Titre: Title:	Conception flexible d'analyses issues d'une trace système
Auteur: Author:	Florian Wininger
Date:	2014
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Wininger, F. (2014). Conception flexible d'analyses issues d'une trace système [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1359/

UNIVERSITÉ DE MONTRÉAL

CONCEPTION FLEXIBLE D'ANALYSES ISSUES D'UNE TRACE SYSTÈME

FLORIAN WININGER
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AVRIL 2014

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

CONCEPTION FLEXIBLE D'ANALYSES ISSUES D'UNE TRACE SYSTÈME

présenté par : WININGER Florian
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. QUINTERO Alejandro, Doct., président
M. DAGENAIS Michel, Ph.D., membre et directeur de recherche
Mme BELLAICHE Martine, Ph.D., membre

C'est le devoir de chaque homme de rendre au monde
au moins autant qu'il en a reçu
Albert Einstein

A toutes les personnes qui m'ont tant donné...

REMERCIEMENTS

Je tiens tout d'abord à remercier mon directeur de recherche, Michel Dagenais, pour la confiance qu'il m'a accordée dès notre première rencontre et qu'il a renouvelée sans cesse depuis lors. Son soutien et sa disponibilité m'ont permis d'en apprendre toujours plus tout en ayant à ma disposition tous les outils me permettant d'avancer.

Je souhaite ensuite remercier les partenaires du laboratoire pour leur soutien financier grâce auquel j'ai pu mener à bien mes travaux. Je remercie également toutes les personnes impliquées dans le projet qui m'ont transmis une partie de leurs connaissances, qui j'en suis sûr m'aideront pour la suite. Un merci tout particulier à Alex, Matthew, Etienne, Francis, Yannick et bien sûr Geneviève pour leur aide précieuse. Merci encore à tous mes collègues du laboratoire DORSAL pour la très bonne ambiance de travail.

J'aimerais remercier mes proches et amis qui ont toujours été là pour moi et m'ont encouragé à me dépasser. Je remercie tout particulièrement mon père, Pascal, pour m'avoir laissé partir sur un autre continent, ainsi que Céline Roehrig et son compagnon Jessy pour m'avoir accueilli à mon arrivée au Canada et m'avoir donné le soutien logistique indispensable à mon installation sur un continent nouveau pour moi.

Enfin, merci à Karine Marle, ma compagne, qui a fait son maximum pour venir me rejoindre au Canada. Un grand merci pour son soutien, ses conseils et sa relecture attentive ainsi que pour tous les bons moments passés ensemble à la découverte du “nouveau monde”.

RÉSUMÉ

Avec la complexité croissante des systèmes informatiques, l'analyse des problèmes de performance devient de plus en plus difficile. L'utilisation des outils de traçage s'est imposée comme la méthode permettant de comprendre en profondeur le fonctionnement d'un logiciel ou d'un système d'exploitation.

Le traceur LTTng (Linux Tracing Toolkit Next Generation) et le visualiseur de traces TMF (Tracing and Monitoring Framework) ont été développés au laboratoire DORSAL de l'École Polytechnique de Montréal. Ce traceur permet de recueillir des traces au niveau du noyau et au niveau des applications utilisateur.

La quantité d'informations sauvegardées dans les traces systèmes est très importante pouvant dépasser les centaines de gigaoctets. L'enjeu principal des techniques d'analyse est d'extraire les informations utiles au développeur pour comprendre son application et l'aider à trouver les anomalies de performance ou de sécurité.

La méthode retenue par le visualiseur de trace TMF pour extraire les données d'une trace est basée sur une machine à états qui permet de modéliser l'état du système à n'importe quel point de la trace. Dans le cas des traces noyaux produites par LTTng, cette méthode permet d'afficher de manière très efficace l'état des fils d'exécution et la consommation de ressources dans des diagrammes en fonction du temps.

Dans ce mémoire, nous proposons de généraliser l'utilisation du gestionnaire d'états de TMF à tous les types de traces, en proposant un outil pour concevoir de manière flexible des nouvelles analyses utilisant une machine à états.

Pour ce faire nous avons créé un langage déclaratif permettant d'utiliser le gestionnaire d'états déjà implémenté dans TMF. La méthodologie suivie dans ce mémoire va permettre de définir un langage dont l'expressivité permettra de caractériser les changements d'états provoqués par les événements de la trace.

Dans un contexte où l'extraction des informations issues d'une trace peut être longue, nous veillons à ce que la performance des nouveaux outils développés soit au moins équivalente à celle des outils actuels. Les résultats obtenus avec l'implémentation de la solution montrent qu'il n'y a pas de dégradation de la performance pour le temps de construction de la machine à états.

Enfin, nous essayons de démontrer l'utilisabilité de la solution en proposant des exemples de nouvelles analyses rendues possibles avec le nouveau langage déclaratif. Ce point passe par l'illustration d'un exemple simple permettant de comprendre la philosophie du nouvel outil. Nous démontrons ensuite la capacité du langage d'abstraire le modèle utilisé, permettant

d'effectuer les mêmes analyses avec des systèmes de traçage différents. Cette étude a permis de transposer le modèle existant du noyau Linux au noyau du système d'exploitation Microsoft Windows.

De même, il a été possible d'enrichir les modèles existants en combinant plusieurs traces issues de différents niveaux d'abstraction, par exemple espace noyau et espace utilisateur. Cela a permis de construire des analyses plus riches, ouvrant la possibilité de corriger des nouvelles classes d'anomalies dans les systèmes étudiés.

Ainsi, le résultat de l'étude montre qu'un langage déclaratif peut être utilisé pour concevoir des analyses flexibles, ne nécessitant aucun développement supplémentaire pour l'utilisateur.

ABSTRACT

With newer complex multi-core systems, performance analysis problems become increasingly difficult. Tracing tools have emerged as a method to understand how a software or an operating system works.

The LTTng tracer (Linux Tracing Toolkit Next Generation) and the TMF trace viewer (Tracing and Monitoring Framework) were developed by the DORSAL laboratory in École Polytechnique de Montreal. This tracer collects traces at both the kernel and user-space levels.

The tremendous amount of data available in traces can exceed hundreds of gigabytes. The main objective for the analysis techniques is to extract the useful information, to help the developer understand his application and find performance or security problems.

The method used by TMF to extract the trace data is based on a state machine, which models the state of the system at any point in the trace. In the case of a kernel trace produced by LTTng, this method allows to show, in a very effective way, the state of threads and the resources consumption in timeline diagrams.

In this work, we propose to generalize the use of the TMF state system to all trace types by providing a tool to design new flexible analyses using a state machine. For this, we created a declarative language to represent the state system already implemented in TMF. This language characterizes the state changes caused by the events in the trace.

In a context where the extraction of information from a trace can take time, we ensure that the performance of new developed tools is at least equivalent to existing tools. The results obtained with the implementation of the proposed solution showed that there is no performance degradation for the construction time of the state machine.

Finally, we try to demonstrate the usability of the solution by providing examples of new analyses made with the new declarative language. At the beginning, we use a simple example to understand the philosophy of the new tool. Then we demonstrate the ability of the new language to abstract the model used to perform the same tests with different tracing systems. This study transposes the existing Linux kernel model to Microsoft Windows kernel.

Similarly, it was possible to extend the existing models by combining several traces from different levels of abstraction, for example kernel space and user space. This helped build more complete analyses. Then it becomes possible to correct new problems in the studied systems.

Thus, the result of the study shows that a declarative language can be used to design flexible analyses without requiring any further development from the user.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

API	Application Programming Interface : Interface de programmation
CTF	Common Trace Format
DTD	Document Type Definition : Définition de type de document
ETW	Event Tracing for Windows
IDS	Intrusion Dectection Systems : Systèmes de détection d'intrusion
LTTng	Linux Trace Toolkit next generation
LTTV	Linux Trace Toolkit Viewer
PDB	Program Database
TMF	Tracing and Monitoring Framework
UST	User-Space Tracer
WPT	Windows Performance Toolkit
XML	Extensible Markup Language : Langage de balisage extensible
XSD	XML Schema Definition : Schéma de définition XML

CHAPITRE 1

INTRODUCTION

Plus que jamais, avec les systèmes multi-coeurs, il est important pour le développeur de comprendre en détail le fonctionnement de son application. Cependant, avec l'ajout de nombreux niveaux d'abstraction, il devient de plus en plus difficile de localiser les anomalies de performance ou de sécurité. C'est pourquoi les logiciels de traçage fournissent des analyses génériques destinées à aider le développeur à trouver la cause de ces problèmes.

En partant du principe que le développeur est la personne qui connaît le mieux son application, nous proposons de définir des analyses flexibles et spécifiques. À partir d'un modèle à états, le développeur devient capable de définir ses propres analyses. Grâce à elles, le développeur peut alors combiner ses connaissances avec l'usage du logiciel d'analyse pour comprendre les causes des problèmes de performance et de sécurité de son application.

Pour parvenir à une telle flexibilité dans la définition des analyses, nous présentons dans ce mémoire un langage déclaratif permettant de créer des modèles à états à partir d'un ensemble d'événements, aussi appelé trace système. Ces modèles permettent de définir et de suivre l'état d'une propriété du système.

1.1 Définitions et concepts de base

Cette section présente les différents concepts et définitions qui seront utilisés tout au long du mémoire. Des définitions additionnelles plus spécifiques au cadre du travail seront ajoutées dans le chapitre 3.

1.1.1 Événement

Dans une trace, un *événement* correspond à une action qui a eu lieu sur le système ou dans l'application. Il atteste le passage du programme par une instruction instrumentée à un instant donné. Par conséquent, un *événement* est un enregistrement *ponctuel* et sa durée dans le temps est nulle.

Un événement est caractérisé par la date à laquelle il est survenu, son type (par exemple un "appel système" dans le noyau ou une entrée de fonction dans une application) et par des informations additionnelles. Ces informations supplémentaires sont propres à chaque type d'événement et renseignent sur le contexte.

1.1.2 État

En opposition à un événement, un *état* possède une durée dans le temps. L'*état* est donc caractérisé à la fois par un *intervalle*, avec un début et une fin, et par sa valeur. Un système complexe peut avoir plusieurs états en même temps.

1.1.3 Attribut

Afin de modéliser l'état complet d'un système, il est nécessaire de diviser cet état en un ensemble de sous-états. La plus petite subdivision du modèle qui puisse avoir un état est appelée *attribut*.

L'*attribut* possède un *unique* état et aide à caractériser un système plus complexe. Un attribut peut être par exemple :

- l'état d'un processeur (actif ou en attente),
- le numéro du processus qui est en train de s'exécuter,
- l'état d'un fichier sur le disque (ouvert, en cours de lecture ou d'écriture, fermé).

Une fois l'attribut créé, il possède nécessairement une valeur d'état disponible pour tous les temps de la trace. Cette valeur peut éventuellement être nulle, comme c'est le cas au début avant sa création effective et à la fin s'il est supprimé. Cette caractéristique assure que l'on puisse faire une requête pour n'importe quel temps de la trace.

1.1.4 Changement d'état

Le *changement d'état* est la transition permettant de passer d'un événement à un état. On définit le *changement d'état* par un temps, un attribut et une valeur d'état.

Le *changement d'état* marque à la fois la fin de l'état précédent (qui peut être nul) et le début d'un nouvel état. Le temps de référence est celui de l'échelle discrète de la trace, à savoir les estampilles de temps données par les événements. La distinction importante entre un *changement d'état* et un événement provient du fait qu'un événement peut provoquer zéro, un ou plusieurs changements d'état.

La valeur du nouvel état est la plupart du temps issue des informations contenues dans l'événement associé. Cependant, il peut également être le fruit d'une combinaison d'autres états déjà présents dans le système, sans pour autant avoir un temps de début ou de fin différent de celui de l'événement qui l'a modifié.

La figure 1.1 montre comment certains événements provoquent des changements d'état à un ou plusieurs attributs. On y voit la trace comme une succession d'événements et l'état global du système comme l'ensemble des attributs du modèle.

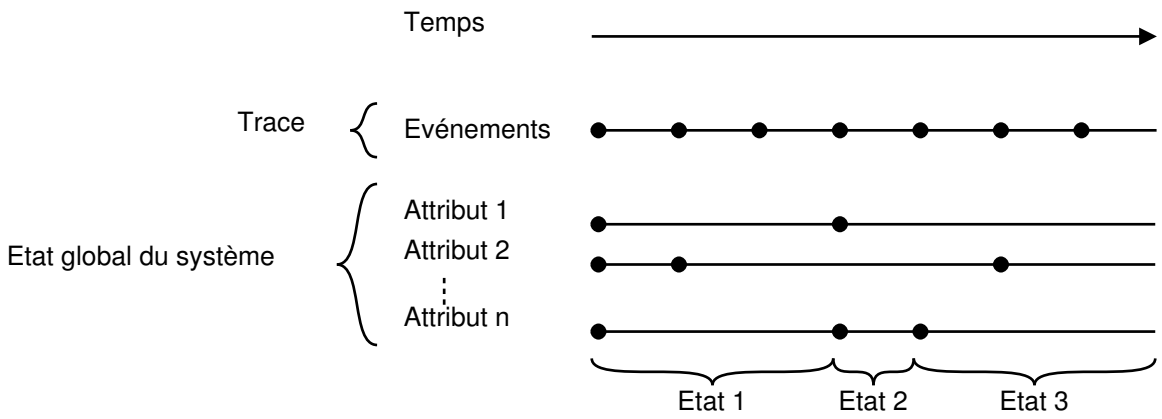

Figure 1.1 Création des intervalles des attributs en fonction du temps.

1.1.5 État global d'un système ou état courant

L'ensemble des attributs à un instant t est appelé *état global du système* ou *état courant*. Il s'agit de l'ensemble des informations que l'on peut déduire de tous les attributs caractérisant le système tracé. L'*état global du système* est l'image exacte d'un système vue par le modèle construit à partir des changements d'état.

L'ensemble des attributs du modèle est organisé à l'aide d'un arbre, appelé **arbre des attributs**. Cet arbre peut être vu comme un système de fichiers, dans lequel fichiers et dossiers sont confondus. Chaque élément de l'arbre, c'est-à-dire chaque attribut, peut contenir un état qui évolue dans le temps.

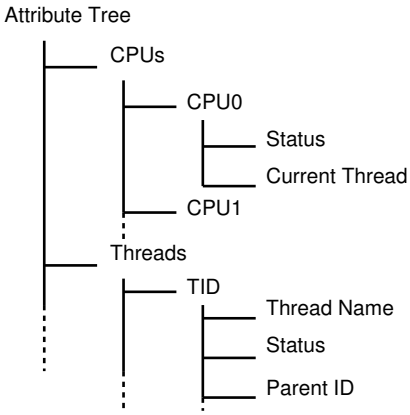

Figure 1.2 Exemple d'arbre des attributs pour une trace noyau.

La Figure 1.2 montre l'arbre des attributs pour une trace noyau, avec notamment les branches `CPUs` et `Threads`. Cet arbre contient l'ensemble des informations permettant de connaître l'occupation des CPUs ou des fils d'exécution des processus, à chaque instant de la

trace.

Il est alors possible de définir sur cette structure un chemin d'accès aux données :

```
/Threads/TID/Status
/CPUs/CPU0/Current Thread
```

Cette structure fournit un moyen d'accès simplifié, qui sera utilisé par les analyses présentées dans le paragraphe suivant, afin d'afficher de manière très efficace les données contenues dans l'arbre des attributs.

1.1.6 Analyse

Afin de rendre à l'utilisateur une information synthétique de sa trace, il est nécessaire de concevoir une *analyse*.

Cette *analyse* est le regroupement entre le modèle qui permet d'extraire les données de la trace, c'est-à-dire la création de l'état global du système au cours du temps, et les informations qui doivent être affichées à l'utilisateur à l'aide de vues génériques.

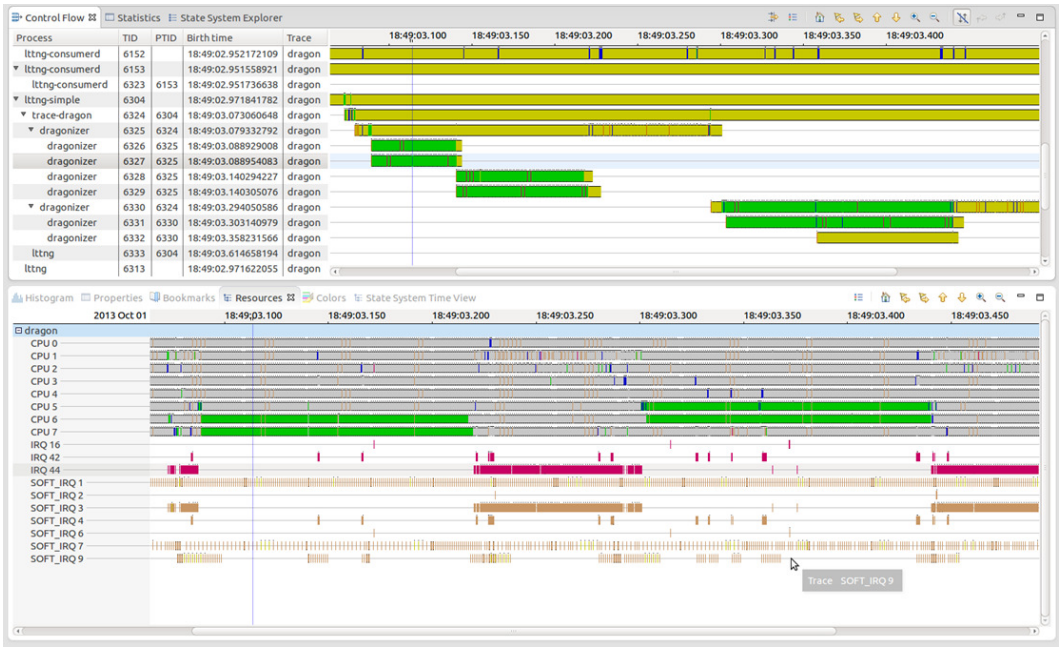

Figure 1.3 Analyse montrant les ressources des processeurs et l'activité des processus pour une trace noyau Linux.

L'intérêt principal de fonctionner avec une *analyse* est qu'il est possible d'optimiser la structure de données utilisée pour extraire les données de la trace afin d'avoir une vitesse d'affichage optimale.

Un exemple d'*analyse* possible est la visualisation de l'occupation des ressources des processeurs, ou de l'activité des processus. La Figure 1.3 montre les deux vues de base pour l'analyse d'une trace noyau Linux. Cette *analyse* repose sur le système de gestion d'état introduit précédemment qui utilise des changements d'états issus des événements noyaux recueillis.

D'autres *analyses* disponibles actuellement sont le calcul du chemin critique d'une application ou encore l'affichage de la pile des appels d'un logiciel.

1.2 Éléments de la problématique

Ces dernières années ont connu un fort développement des libraires, des cadres d'application (*framework*), de la virtualisation et de l'informatique nuagique. Toutes ces innovations technologiques permettent de créer des nouvelles applications toujours plus complexes, et surtout sans reproduire du code déjà existant. Elles introduisent toutefois des niveaux d'abstractions supplémentaires et éloignent toujours plus le développeur de logiciel du matériel.

Dans le même temps, les composants matériels deviennent toujours plus sophistiqués. D'un côté, les ordinateurs personnels sont devenus fortement multi-processeurs pour gagner en vitesse, obligeant le développeur à concevoir des applications parallèles pour tirer partie de ces architectures. De l'autre côté, on voit la forte expansion des périphériques mobiles aux ressources plus limitées, mais souhaitant accueillir des applications toujours plus complexes.

Dans ce contexte, trouver les causes premières d'un problème de performance ou de sécurité peut devenir rapidement très difficile. Pour cette raison, des outils de traçage ont vu le jour afin d'aider le développeur ou l'administrateur système à comprendre et à identifier ces problèmes. Ces outils permettent de répondre à des problématiques spécifiques en proposant des analyses permettant de trouver des problèmes typiques.

Cependant, il reste beaucoup d'enjeux dans la conception d'analyses issues de traces systèmes. Les analyses standards proposées ont souvent des limitations et la création de nouvelles analyses est souvent fastidieuse pour l'utilisateur qui n'est pas un spécialiste du logiciel d'analyse et n'a pas envie de passer du temps à programmer ce dernier. Il serait donc bon pour les logiciels d'analyse de trace de permettre à l'utilisateur d'ajouter facilement de l'information dans les analyses.

Par ailleurs, plus les données produites par les traceurs sont précises, plus la taille des traces, c'est-à-dire le nombre d'événements, devient important. Il n'est pas surprenant de créer des centaines de gigaoctets de fichiers de traces pour effectuer une analyse détaillée. Les outils d'analyse doivent donc construire des modèles particulièrement efficaces pour extraire les informations pertinentes dans cet océan de données. C'est pourquoi l'extraction d'informations précises passe par la modification du modèle pour l'adapter au contexte. Ainsi, la définition d'analyses flexibles par l'utilisateur aide à augmenter l'efficacité des logiciels d'analyse de trace.

1.3 Objectifs de recherche

La question de recherche qui découle de la problématique est la suivante :

> L'introduction d'un langage descriptif pour exprimer les relations entre les événements de la trace et l'état d'un système permet-elle de concevoir simplement des analyses et de résoudre des nouveaux problèmes de performance et de sécurité ?

Il est maintenant possible de préciser les objectifs de recherche :

1. Caractériser de manière formelle les transitions d'états d'un système provoquées par les événements d'une trace.
2. Développer un langage descriptif pour définir de manière générique les transitions d'états permettant de modéliser le système étudié.
3. Valider que la solution conserve ou améliore les critères de performance des analyses.
4. Déterminer les possibilités d'adaptation de la solution sur différents systèmes (Linux ou Windows) et à différents niveaux (Noyau ou Application)

L'ensemble de ces objectifs a pour but de répondre à l'objectif principal de généraliser les outils existants pour faire le lien entre les informations contenues dans une trace et l'état résultant d'un système modélisé, afin de concevoir des analyses flexibles.

1.4 Plan du mémoire

Au chapitre 2, nous présenterons une revue de littérature faisant le point sur l'état de l'art dans les domaines du traçage, de la modélisation de l'état d'un système, de la visualisation d'une trace et des langages de définition.

Le chapitre 3 présente les aspects de la méthodologie de cette étude afin d'introduire l'article "Declarative Specification for Constructing the Modeled State from a System Execution Trace" du chapitre 4. Cet article présente l'essentiel des travaux de recherche réalisés pour répondre à la problématique de la section 1.2. Suit la discussion complémentaire au chapitre 5 qui étend les résultats de l'article. Enfin, le chapitre 6 fera une synthèse des travaux en identifiant les limites et les évolutions futures possibles.

CHAPITRE 2

REVUE DE LITTÉRATURE

Ce chapitre présente l'état de l'art dans les domaines du traçage et de la modélisation d'un système par une approche par état. Il se divisera en quatre sujets relatifs à la problématique présentée précédemment.

Le premier thème traitera du traçage en général. Il présentera les différentes façons de générer des traces d'événements. Le second sujet concernera la modélisation de l'état d'un système informatique à partir des traces. Ce sera notamment l'occasion de présenter le principe de fonctionnement des logiciels développés au laboratoire DORSAL, puisque le projet repose sur les travaux et les développements antérieurs. Le troisième sujet montrera l'état de l'art de la façon de visualiser graphiquement les informations d'une trace. Enfin, la quatrième partie traitera des langages de modélisation. Cette partie permettra de voir quels sont les langages existants pour notre type d'application et quelles sont leurs limites.

Nous informons le lecteur que tout le travail présenté dans cette revue de littérature n'utilise que des algorithmes publiés et des logiciels libres, mise à part la partie sur Event Tracing for Windows (ETW) de la suite Windows Performance Toolkit (WPT).

L'avantage majeur de travailler avec des logiciels libres est d'avoir accès au code source sans aucune restriction afin de bien comprendre leur fonctionnement. Dans le cas de ETW, nous baserons nos propos sur les informations fournies par Microsoft qui malheureusement ne sont pas toujours très précises sur les algorithmes utilisés.

2.1 Le traçage

Dans un premier temps nous présenterons un état de l'art des solutions de traçage sur le système d'exploitation Linux. Grâce à l'ouverture de cette plateforme, les nombreuses avancées technologiques y sont intégrées et permettront de couvrir la littérature dans ce domaine. Nous finirons cette section par la présentation de la solution de traçage propriétaire de Microsoft pour sa plateforme Windows.

2.1.1 Généralités

Le traçage est une technique d'analyse d'anomalies des systèmes informatiques. En récupérant des informations sur une application durant son exécution, tout en essayant de limiter au maximum l'impact sur cette dernière, il est possible d'observer son comportement sans

l'interrompre (contrairement au débogage). Nous pouvons ainsi obtenir une vue d'ensemble d'un système en cours d'exécution, afin de détecter les sources des différents problèmes de performance ou de sécurité.

Il existe deux approches différentes pour le traçage : l'instrumentation dynamique et l'instrumentation statique.

L'instrumentation dynamique

Comme son nom l'indique, ce type d'instrumentation permet d'insérer dynamiquement à l'endroit désiré un dispositif équivalent à un point d'arrêt ainsi que du code permettant de traiter automatiquement le point d'arrêt et de reprendre l'exécution du programme là où il a été suspendu.

L'avantage de la méthode est que les points de trace peuvent être ajoutés sans à avoir à recompiler le code du programme. Cependant, le surcoût important de la méthode rend son utilisation plus difficile car elle perturbe plus l'environnent du système en cours d'exécution. Sous le noyau Linux, l'instrumentation se fait à l'aide de **kprobes** (voir Goswami, 2005).

L'instrumentation statique

L'instrumentation statique utilise quant à elle des points de trace ajoutés de manière fixe dans le code source de l'application, ce qui nécessite d'avoir le code source et de pouvoir recompiler l'application. Ces points de trace sont ajoutés par la macro `TRACE_EVENT()`. Rostedt (2010a,b,c) fait une analyse approfondie de l'utilisation de cette macro. `TRACE_EVENT()` a vu le jour afin d'améliorer la flexibilité et de donner aux développeurs un mécanisme standard pour ajouter des points de trace.

L'avantage principal de cette macro est qu'elle est indépendante du traceur qui va s'y connecter. Il est donc possible pour le développeur d'ajuster le code de la macro de façon à optimiser le code qui va être généré à la compilation. Dans le cas du noyau Linux, les traceurs tels que *perf, ftrace, SystemTap* ou encore *LTTng* sont capables de l'utiliser. D'ailleurs, le même type de macro est utilisé dans *Google Chromium* pour instrumenter le code, ce qui permet d'ajouter facilement un nouveau point de trace.

2.1.2 Le traçage au niveau noyau

Dans cette partie, nous allons voir en particulier des traceurs utilisés sur le noyau Linux. La particularité du traçage sur le système d'exploitation Microsoft Windows sera traitée séparément.

Le traceur SystemTap

Corbet (2008) fait un état de l'art des systèmes de traçage sous Linux. Il présente notamment **SystemTap**. Cette solution de traçage a été développée depuis 2005 par Red Hat. Ce traceur vise principalement les administrateurs système en proposant un système de surveillance pour Linux (voir Eigler et Red Hat, 2006). Il permet d'ajouter de manière simplifiée des points de traces dynamiques via *kprobes*. Ce traceur met l'accent sur sa flexibilité d'utilisation en utilisant un langage de script. Cependant, l'accent n'est pas mis sur la performance et le surcoût engendré par ce traceur est important.

Il ne pourra pas être utilisé dans le cadre de notre étude car l'analyse des résultats se fait à la volée et est affichée directement dans une console ou écrite dans un fichier. Il n'est pas possible de sauvegarder efficacement les événements bruts pour concevoir une analyse par dessus, comme nous le faisons dans notre étude.

Le traceur ftrace

La traceur **ftrace**, pour *function tracer*, est un traceur intégré au noyau linux créé avec l'objectif de suivre l'ordre d'appel des fonctions dans le noyau. Une introduction à son fonctionnement est proposée par Ficheux (2011) et par Rostedt (2008).

ftrace utilise la macro `TRACE_EVENT()` présente dans le noyau. Ce traceur est capable de produire des informations concernant les latences, le traitement des interruptions et les changements de contexte. Il permet également de générer des graphes d'appels de fonctions.

Le traceur perf

Une autre façon de suivre l'activité d'un système est de suivre l'activité des compteurs de performance, tels que les changements de contexte, les fautes de page ou encore les fautes de cache.

Le traceur **perf** a été intégré au noyau Linux en 2009 avec pour objectif d'accéder facilement aux différents compteurs de performance présents sur les processeurs (voir Edge, 2009). Par la suite, une extension a permis d'utiliser la macro `TRACE_EVENT()` pour lire les points de trace du noyau Linux. Il devient ainsi un traceur noyau complet et performant, notamment grâce à l'utilisation de tampons circulaires.

Les analyses issues de **perf** sont sous la forme de statistiques car le traceur fonctionne par échantillonnage de valeurs. C'est-à-dire qu'à chaque fois que l'on dépasse une limite choisie, une interruption est générée sur le système de sorte à pouvoir enregistrer la valeur du compteur. Ce type d'analyse permet d'avoir un plus faible impact sur le système, mais les données obtenues sont moins précises.

Le traceur LTTng

Le traceur **Linux Trace Toolkit next generation (LTTng)** a été créé pour réaliser l'instrumentation du noyau en essayant de minimiser l'impact sur les performances du système, afin de permettre de tracer des systèmes fortement multiprocesseurs (voir Desnoyers et Dagenais, 2006; Desnoyers, 2009).

Dans la version initiale de LTTng `0.X`, le traceur était un ensemble de correctifs à appliquer aux sources du noyau Linux pour ajouter des points de trace non standards. Depuis la version `2.0`, sortie en 2012 (voir Desnoyers *et al.*, 2012), la partie traceur est sous la forme de modules noyau qui sont chargés au moment du démarrage de l'outil de traçage et qui utilisent les différents points de trace statiques présents à l'aide de la macro `TRACE_EVENT()`. **LTTng** permet également d'enregistrer les compteurs de performance et d'activer des points de trace dynamiques utilisant *kprobes*.

Lors du traçage, les événements produits sont écrits dans des tampons circulaires. Un processus externe, appelé consommateur, se charge de vider ces tampons en les écrivant sur le disque dur. L'utilisation de tampons permet de garantir la mise à l'échelle et ne nécessite pas d'attente. Les variables de contrôle pour ces tampons sont mises à jour par des opérations atomiques plutôt que par des verrouillages.

Le format de sortie des événements est le Common Trace Format (CTF), un format libre (voir Desnoyers et EfficiOS Inc, 2013). Ce format de trace est binaire et par conséquent très compact, pour une performance optimale lors de l'écriture sur disque. Grâce à un fichier de métadonnée, il est possible de décoder les événements pour en faire l'analyse ultérieurement. En particulier, il peut être lu en mode console par l'utilitaire **babeltrace** fourni dans la suite d'utilitaires LTTng.

Les événements lus par babeltrace sont de la forme suivante :

```
[11:24:42.469766615] (+0.000001396) softirq_entry: { cpu_id = 0 }, { vec = 9 }
[11:24:42.469768012] (+0.000000698) softirq_exit: { cpu_id = 0 }, { vec = 9 }
[11:24:42.469773460] (+0.000000699) sched_switch: { cpu_id = 0 },
        { prev_comm = "swapper", prev_tid = 0, prev_prio = 20, prev_state = 0,
          next_comm = "lttng-consumerd", next_tid = 4085, next_prio = 20 }
[11:24:42.469776882] (+0.000002794) exit_syscall: { cpu_id = 0 }, { ret = 1 }
```

2.1.3 Le traçage au niveau application

Dans cette partie nous allons regarder les différentes façons de tracer une application. Cette méthode, aussi nommée traçage en espace utilisateur, repose sur le même principe que

le traçage au niveau du noyau. Il s'agit de recueillir cette fois des événements au sein d'une application, sans interrompre son exécution, tout en limitant l'impact sur celle-ci.

Une façon primitive de faire du traçage au niveau d'une application est d'utiliser la sortie console standard à l'aide de la fonction `printf()`. Cette approche se révèle toutefois complètement inefficace. Comme pour le traçage au niveau du noyau, plusieurs approches plus efficaces ont été développées.

Le traceur DTrace

Le traceur **DTrace** est le traceur de référence sur le système d'exploitation Solaris (voir Cantrill *et al.*, 2004). Ce traceur est reconnu pour ne causer pratiquement aucun impact de performance sur le système si le traçage est désactivé, contrairement aux traceurs qui utilisent des points de trace statiques.

De plus, il offre la capacité de tracer à la fois en mode noyau et en mode utilisateur. Il propose une facilité d'intégration dans la plupart des langages classiques et des langages de script (voir Rajadurai, 2012).

Le traceur SystemTap

SystemTap peut également être utilisé pour tracer des applications. Avant le noyau Linux 3.8, il était nécessaire d'ajouter un correctif au niveau du noyau appelé *utrace*. Il contient un code source qui n'est pas intégré au noyau et qui remplace l'appel système *ptrace* de façon transparente pour l'utilisateur, afin d'utiliser les fonctionnalités de traçage en espace utilisateur. Depuis le noyau 3.8, l'extension *uprobes* est disponible directement dans le noyau. *uprobes* pour User space Probes fonctionne de la même manière que *kprobes* mais en espace utilisateur (voir Keniston et Dronamraju, 2010).

Cependant tracer une application avec **SystemTap** reste relativement plus coûteux que d'autres solutions, car un appel système est réalisé pour chaque événement capturé. Un tel comportement provoque un impact majeur sur la performance de l'application qui est instrumentée.

Le traceur LTTng-UST

User-Space Tracer (UST) est la solution du traceur LTTng pour tracer des applications en espace utilisateur (voir Fournier *et al.*, 2009). Cette extension fournit des macros permettant d'ajouter des points de trace statiques dans le code d'un programme.

Contrairement à *SystemTap*, **LTTng-UST** évite de faire un appel système pour chaque événement. L'application tracée se charge d'écrire dans de la mémoire partagée, qui peut

ensuite être lue par un autre processus, le consommateur de traces. Le consommateur est spécifique pour les traces d'application et il est différent de celui du noyau. L'application écrit directement les événements en mémoire. Ainsi, elle n'a pas besoin de passer par le noyau du système d'exploitation. Une fois le tampon de mémoire rempli, l'application prévient le consommateur par un tube de contrôle non bloquant. Ce dernier se charge d'écrire le tampon sur disque pendant que l'application continue de s'exécuter et de produire des événements dans un autre tampon. De plus, afin d'offrir une plus grande mise à l'échelle, UST permet de partager les mêmes tampons pour les différents processus tracés.

Le format de sortie est toujours le CTF détaillé précédemment.

Le traceur intégré à Google Chromium

Le navigateur Google Chromium présente une autre manière de tracer au niveau de l'application. En effet, ce logiciel possède un système de traçage embarqué (Chromium Project, 2013a). Ce système repose sur différentes macros `TRACE_EVENT()` permettant de rediriger les événements vers des applications externes ou un visualiseur interne [1] (voir Chromium Project, 2013b).

Ce projet montre une solution complète de traçage au niveau application. L'avantage de cette technique est la portabilité. En effet, elle fonctionne sur toutes les versions du logiciel, quel que soit le système d'exploitation.

Cependant, la portabilité de ce système, due à son interface Java Script et à son recours au traçage en mémoire, est aussi une limite de la solution par rapport à sa capacité à recueillir et à afficher de grandes traces. En effet, contrairement à des traces écrites sur disque, les traces de Chrome sont limitées à quelques secondes et ne contiennent qu'un petit nombre d'événements.

La vue présentée à la Figure 2.1 permet de montrer la pile d'appels des fonctions du navigateur en fonction du temps.

Par ailleurs, de récents développements ont intégré des sources plus larges comme des événements noyaux issus d'ETW sous Windows. Il est alors possible d'ajouter les appels systèmes dans la pile d'exécution pour enrichir la vue.

De plus, un autre outil de traçage semblable a été développé pour tracer le système d'exploitation Android. Il permet d'afficher l'usage des processeurs en plus de l'utilisation des différents fils d'exécution (voir Android Project, 2014).

1. chrome ://tracing

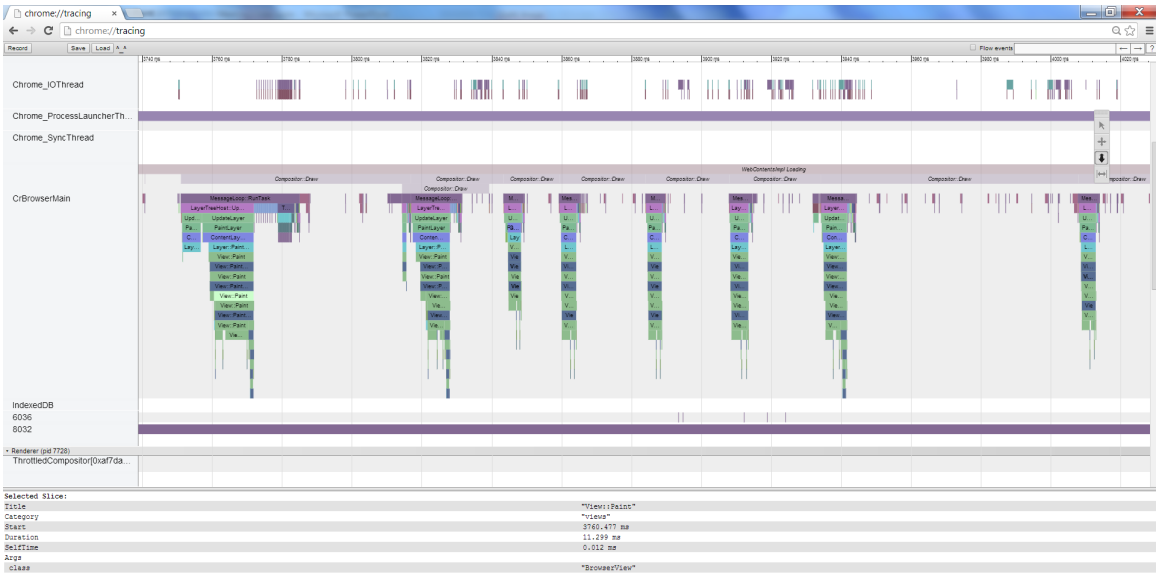

Figure 2.1 Vue de type Gantt du visualiseur de trace de Chromium.

2.1.4 Le traçage sous Windows

Introduit avec Windows 2000, ETW est une plateforme générique de traçage pour le système d'exploitation de Microsoft. En utilisant un principe efficace de tampons et un mécanisme d'instrumentation dans le noyau, ETW fournit un système permettant de recueillir des événements à la fois en mode noyau et en mode utilisateur.

Le système présenté par Microsoft (voir Park et Buch, 2007) est flexible, mais également robuste. Il permet à n'importe quelle application de devenir un fournisseur d'événements à l'aide d'une Interface de programmation (API). De plus, il propose un format de spécification des événements, le MOF [2], permettant à l'utilisateur d'ajouter et d'échanger ses propres types d'événements.

Lors de la capture d'une trace, ETW génère automatiquement un cliché de l'état courant du système avec tous les processus et fils d'exécution actifs au démarrage et à l'arrêt, facilitant ainsi le démarrage d'une analyse par machine à états.

Le traceur au niveau noyau est capable de fournir tous les événements relatifs aux processeurs, à l'ordonnanceur et à la mémoire et bien d'autres encore, disponibles sur le site de référence MSDN [3].

Une seconde force de ETW est sa capacité à utiliser le système d'information de débogage

2. MSDN Providing Events
3. MSDN Event Tracing

pour les modules exécutables appelés Program Database (PDB). Ce système permet de faire la résolution automatique des noms en téléchargement sur les serveurs de mises à jour des PDB associés aux différents exécutables et librairies du système d'exploitation (voir Park et Bendetovers, 2009a,b). Il est alors possible d'avoir une bonne performance en stockant dans la trace uniquement les adresses des fonctions, tout en étant capable de retrouver leur nom au moment de l'ouverture de la trace.

2.2 Modélisation de l'état d'un système

Pour compléter les solutions de traçage, il existe un grand nombre de visualiseurs de traces permettant d'extraire l'information et de l'afficher à l'utilisateur. En séparant le traçage de l'analyse, il est possible d'avoir un traçage avec le plus faible impact possible sur le système et de faire les tâches coûteuses d'analyse a posteriori.

Puisque les travaux de recherche menés s'intègrent au projet LTTng, nous nous intéressons dans un premier temps à la méthode d'analyse basée sur un gestionnaire d'état utilisé par Linux Trace Toolkit Viewer (LTTV) [4] et Tracing and Monitoring Framework (TMF) [5]. Nous verrons ensuite les alternatives, notamment de stocker directement l'information d'état dans la trace.

2.2.1 La méthode du gestionnaire d'état

Comme nous l'avons dit dans l'introduction, une méthode couramment utilisée par les visualiseurs de trace est de modéliser le système avec une approche par états. La création d'une machine à états, créée à partir d'événements de la trace, permet de pouvoir faire des requêtes sur l'état du système à n'importe quel point de la trace. Une étude détaillée sur l'extraction d'informations à partir d'une trace noyau a été réalisée (voir Giraldeau *et al.*, 2011).

Les visualiseurs de trace comme TMF ou LTTV utilisent un modèle d'état pour afficher les informations d'ordonnancement des processeurs et des processus sous la forme d'un diagramme de Gantt. Ce modèle fonctionne pour les traces noyau Linux issues du traceur LTTng. La définition du modèle d'état, c'est-à-dire les *changements d'état*, est faite à même le code du visualiseur.

La création de la machine à états se fait au moment de l'indexation de la trace en lisant la trace depuis le début, dans l'ordre, et sans retour possible. Le gestionnaire d'état débute avec des attributs contenant des états vides. Il y a donc un période durant laquelle la machine n'est

4. http ://lttng.org/lttv
5. http ://www.eclipse.org/linuxtools/projectPages/lttng/

pas initialisée et comprend des informations incomplètes. Par exemple, il n'est pas possible de connaître quel processus est exécuté sur un processeur donné avant d'avoir eu un événement de l'ordonnanceur (`sched_switch` avec LTTng).

On peut voir sur la Figure 2.2 l'évolution du statut d'un thread à son initialisation. Avant le premier `sched_switch`, l'attribut n'existe pas. Par la suite il change d'état en fonction des événements qui lui sont associés.

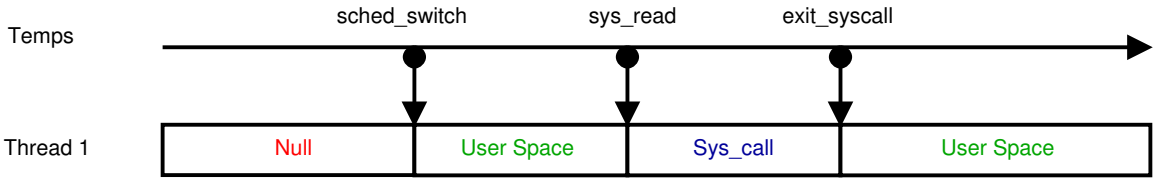

Figure 2.2 Exemple de conversion d'événements en états.

De même, un autre problème lié au fonctionnement par changement d'état est la mauvaise résilience du gestionnaire d'état par rapport à la perte d'événements. Pour la même raison que dans l'initialisation, en cas de perte d'un événement, on reste dans un état incorrect pendant une durée indéterminée. La Figure 2.3 montre le cas de la perte de l'événement `sys_read` qui devrait changer le statut du thread en "fait un appel système". Dans cette situation, l'état du fil d'exécution n'est pas correct jusqu'au prochain événement. Il n'existe pour le moment aucun mécanisme permettant d'afficher à l'utilisateur que l'état n'est pas certain. Il revient donc à l'utilisateur de faire attention à la façon dont sont recueillies les traces et de veiller à ne pas avoir d'événements perdus.

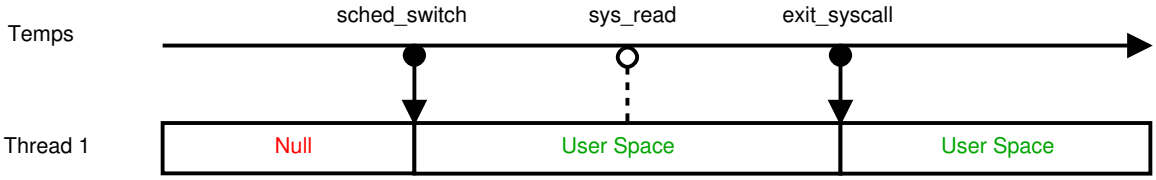

Figure 2.3 Exemple des conséquences d'une perte d'un événement sur la machine à états.

Pour aider à construire une machine à états, le traceur ETW propage une série d'événements au démarrage et à la fin de la trace, permettant d'initialiser l'état des différents processus. LTTng propose également ce système via l'événement `lttng_statedump_process_state`. De plus, les deux traceurs sont capables d'indiquer le nombre d'événements perdus afin de déterminer la fiabilité des informations contenues dans la trace.

2.2.2 Stocker l'information d'état dans une structure dédiée

Compte tenu du coût de construction de la machine à états, une méthode inefficace serait de devoir relire la trace depuis le début pour chaque requête faite dans le gestionnaire d'état. Une solution retenue par le visualiseur TMF est de stocker ce fichier, qui est de l'ordre de grandeur de la taille de la trace sur disque, à l'aide d'une structure d'arbre permettant de faire rapidement des requêtes par la suite, en profitant du critère de recherche logarithmique d'un arbre de recherche (voir Montplaisir-Goncalves *et al.*, 2013). Cette structure de données est capable de supporter des traces de plusieurs dizaines de Gigaoctets.

L'hypothèse utilisée pour cette structure de données est que les intervalles d'état arrivent par ordre croissant de temps de fin. Cela permet de garantir un critère de recherche selon le temps sans devoir rééquilibrer l'arbre pendant sa création et de l'écrire directement sur le disque de façon continue.

Une extension du système a également été proposée pour diminuer l'espace disque utilisé. Cette solution propose de ne pas sauvegarder chaque intervalle d'état, mais d'enregistrer à la place des clichés, ou *snapshots*, de l'arbre d'état à des durées fixées. Avec cette méthode, il est possible de retrouver l'état d'un attribut pour n'importe quel temps en se positionnant au cliché précédent et en reconstruisant la machine à états jusqu'au temps désiré. Avec un compromis de performance au niveau de la vitesse des requêtes, il est alors possible de réduire la taille du fichier stocké d'un facteur de plus de 100.

2.2.3 Stocker l'information d'état dans la trace

Une seconde manière de stocker l'information d'état est de le faire directement dans la trace. Cette approche est utilisée par Chan *et al.* (2008). Leurs traces contiennent à la fois des événements ponctuels, et des états avec une durée.

Néanmoins, cette approche présente l'inconvénient que les informations d'état issues du traceur sont très étroitement liées avec l'affichage, ce qui donne moins de flexibilité. Ce choix n'est pas possible dans le cas de TMF qui souhaite être un visualiseur acceptant tous les formats de traces et ne souhaite pas être intégré à un traceur particulier. C'est d'ailleurs ce choix qui va nous permettre d'ouvrir les analyses à base d'état à d'autres plateformes dans le chapitre 4.

De plus, il est techniquement possible avec LTTng d'ajouter l'information d'état au moment du traçage. Cependant ce traceur est présenté comme un traceur haute-performance, qui affecte très peu le comportement d'un système instrumenté. Le surcoût du calcul et du stockage des états est donc laissé au visualiseur de trace, lors de la phase d'analyse.

2.3 Visualisation d'une trace

Maintenant que nous avons vu comment recueillir une trace et comment stocker les informations pertinentes pour l'utilisateur, nous allons nous intéresser dans cette partie aux différentes manières de représenter graphiquement le contenu d'une trace.

Actuellement, plusieurs outils permettent de visualiser et d'analyser des traces. Nous allons détailler les principales formes de visualisation existantes.

2.3.1 Visualisation brute

Nous avons vu dans la partie sur les traceurs qu'il est possible de lire une trace en mode texte. Des logiciels tels que Babeltrace permettent d'afficher en console le contenu de la trace.

Il existe également un outil permettant de visualiser en mode texte les informations d'un modèle à états dans TMF (voir Montplaisir *et al.*, 2013). Cet outil, présenté à la Figure 2.4, permet d'afficher à un temps choisi le contenu complet de l'arbre des attributs du modèle à états.

En plus de l'information contenue dans l'attribut, on obtient les détails complets de l'état, à savoir son début, sa fin et sa valeur. Cette vue permet également de filtrer pour afficher uniquement les changements d'état ayant eu lieu à l'instant affiché. Nous connaissons alors l'impact d'un événement sur le modèle, ce qui facilite sa validation.

Histogram | Properties | State System Explorer

State System / Attribute	Quark	Value at timestamp	Start time	End time	Full attribute path
CPUs	0		11:24:42.440 133 097	11:24:52.664 579 801	CPUs
1	1		11:24:42.440 133 097	11:24:52.664 579 801	CPUs/1
0	16		11:24:42.440 133 097	11:24:52.664 579 801	CPUs/0
Current_thread	17	2290	11:24:42.442 235 251	11:24:42.442 407 549	CPUs/0/Current_thread
Status	25	2	11:24:42.442 342 039	11:24:42.442 346 647	CPUs/0/Status
Threads	3		11:24:42.440 133 097	11:24:52.664 579 801	Threads
-1	4		11:24:42.440 133 097	11:24:52.664 579 801	Threads/-1
4084	8		11:24:42.440 133 097	11:24:52.664 579 801	Threads/4084
4087	9		11:24:42.440 133 097	11:24:52.664 579 801	Threads/4087
4072	18		11:24:42.440 133 097	11:24:52.664 579 801	Threads/4072
0	19		11:24:42.440 133 097	11:24:52.664 579 801	Threads/0
4085	28		11:24:42.440 133 097	11:24:52.664 579 801	Threads/4085
4088	33		11:24:42.440 133 097	11:24:52.664 579 801	Threads/4088
4089	38		11:24:42.440 133 097	11:24:52.664 579 801	Threads/4089
2290	48		11:24:42.440 133 097	11:24:52.664 579 801	Threads/2290
Status	49	3	11:24:42.442 342 039	11:24:42.442 346 647	Threads/2290/Status
System_call	58	sys_read	11:24:42.442 342 039	11:24:42.442 346 647	Threads/2290/System_call
Exec_name	59	compiz	11:24:42.441 770 736	11:24:52.664 579 801	Threads/2290/Exec_name
PPID	60		11:24:42.440 133 097	11:24:42.490 492 646	Threads/2290/PPID

Figure 2.4 Vue de l'état global du système

2.3.2 Les diagrammes de Gantt

L'une des vues les plus courantes pour représenter l'ordonnancement et les ressources est le diagramme de Gantt. Issu de la gestion de projet, ce type de diagramme permet de visualiser efficacement dans le temps les diverses "tâches" d'un projet.

Dans notre contexte, ce diagramme est utilisé pour afficher l'état d'une liste d'attributs caractéristiques du système tracé. Avec le logiciel TMF, cette vue est fortement corrélée avec le contenu du gestionnaire d'état, ce qui permet d'avoir des vitesses d'affichage optimales car il n'y a plus d'information supplémentaire à calculer.

La plupart des logiciels proposent ce type de vue. La Figure 2.5 montre l'utilisation qui en est faite par le visualiseur développé par Microsoft pour lire des traces ETW. Cette vue permet ici de représenter l'activité des processus et l'utilisation du disque dur.

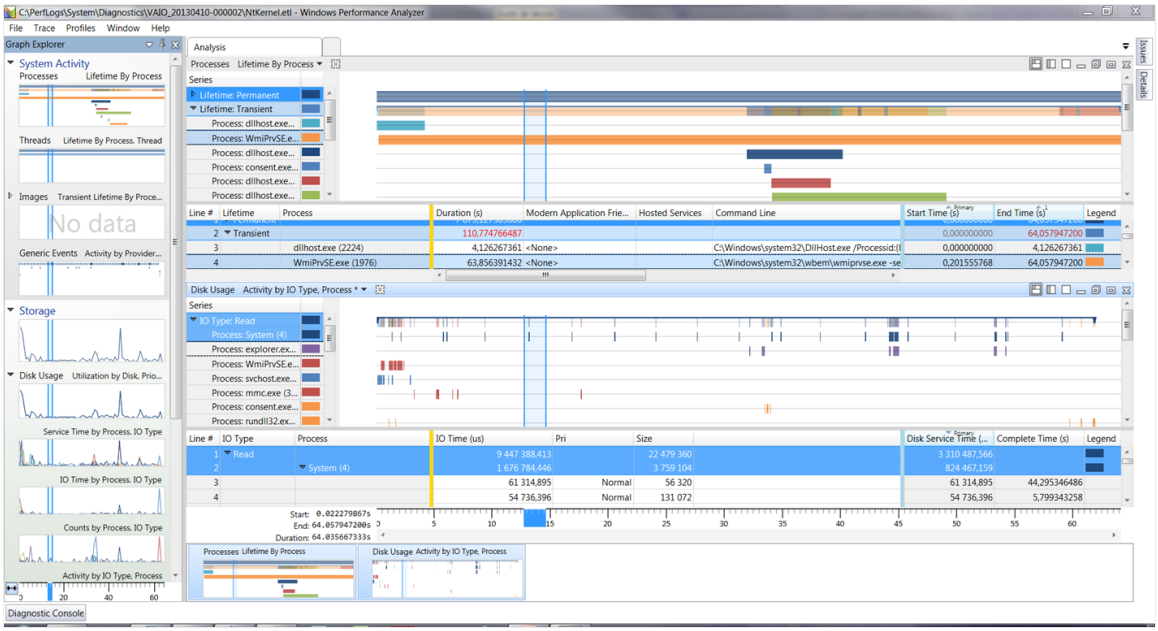

Figure 2.5 Logiciel de visualisation de Windows Performance Analyzer.

Le logiciel Jumpshot permet d'utiliser cette même représentation pour afficher directement les états contenus dans la trace (voir Chan *et al.*, 2008) . L'utilisation principale de ce logiciel est de pouvoir analyser des exécutions de programmes utilisant la librairie MPI (voir Zaki *et al.*, 1999; Gropp *et al.*, 1999).

Des recherches au niveau de l'abstraction des données pour réduire le nombre d'états affichés ont été réalisées par Ezzati-Jivan et Dagenais (2012). Suivant le principe qui est mis en place pour l'affichage de labels pour les données géographiques, il est proposé dans cette solution d'afficher des états synthétiques regroupant plusieurs états, pour augmenter la

lisibilité suivant le niveau de zoom de la vue. Ainsi, par exemple, les étapes pour faire une requête Internet vers un serveur `http` peuvent être vues au plus bas niveau comme l'ensemble des états *requête DNS*, *établissement de la connexion TCP*, *envoie de la requête HTTP*. On peut ensuite les regrouper à plus haut niveau comme un seul état *connexion HTTP*.

2.3.3 Graphiques

Les graphiques fonctionnels sont un second moyen de visualiser des données de façon synthétique en fonction du temps. Le plus souvent, ces graphiques utilisent le temps comme abscisse et permettent donc de représenter l'évolution d'un paramètre, c'est-à-dire un attribut. Il existe plusieurs représentations équivalentes parmi lesquelles les courbes, les histogrammes ou encore les diagrammes à barres.

TMF et Jumpshot offrent une vue en histogramme pour montrer la densité d'une information, comme le nombre d'événements. Une vue utilisant une courbe est également disponible pour afficher la consommation mémoire durant le temps d'exécution.

2.3.4 Graphes de relations

Un nouveau moyen d'explorer le contenu d'une trace est l'utilisation de graphes pour mettre en relation les données de façon structurée. Le logiciel d'analyse Triva [6] de l'INRIA permet de visualiser de façon interactive des traces d'exécution d'applications parallèles (voir Schnorr *et al.*, 2010). Le logiciel utilise les messages de synchronisation entre les différents fils d'exécutions parallèles pour construire un graphe à trois dimensions. Les deux dimensions du plan sont utilisées pour établir le graphe, puis la troisième permet d'ajouter la dimension temporelle. Il présente la façon de passer d'une vue de Gantt à cette représentation.

Ce graphe permet d'avoir les mêmes informations que le diagramme de Gantt. Cependant, elle ajoute un niveau supplémentaire de visibilité pour l'utilisateur en présentant différemment les interactions entre les attributs, i.e. les lignes du précédent diagramme de Gantt.

2.3.5 Statistiques

Une autre façon de représenter les données d'une trace est de générer une table de statistiques. Cette approche est utilisée pour l'analyseur de performances de base de Windows (sans le Windows Performance Toolkit). Il fournit un grand nombre de métriques issues des compteurs de performance et des points de trace du noyau. Ces données comportent souvent une dimension temporelle incluse, comme une fréquence. Elles sont fournies avec une moyenne, un minimum et un maximum. On y retrouve notamment :

6. http ://triva.gforge.inria.fr/

– le nombre d'interruptions par seconde des processeurs,
– le nombre de fautes de pages par seconde,
– le nombre d'octets lus et écrits sur disque par seconde,
– les fichiers les plus utilisés,
– le pourcentage de temps passé en utilisateur ou en appel système,
– le nombre d'octets émis et reçus du réseau...

Une autre forme de statistiques possible est l'utilisation d'une trace pour générer l'équivalent du programme `top`[7] sous Linux. Ce programme permet d'afficher les processus s'exécutant actuellement sur les processeurs ainsi que les ressources qui lui sont allouées (quantité de mémoire, pourcentage du CPU...). Le logiciel *lttngTop* de la suite des outils de LTTng offre les mêmes fonctionnalités en utilisant les informations recueillies par une trace noyau LTTng (voir Desfossez, 2011). Des mises à jours récentes permettent d'utiliser ce logiciel en direct (*live mode*) et sur le réseau.

Si la représentation de statistiques à l'aide d'un tableau semble la plus évidente, Triva permet également de représenter des données sous la forme d'un graphique de type *treemap* (voir Schnorr *et al.*, 2009).

Une étude sur la façon de calculer des statistiques à partir de grands fichiers de traces a été faite par Ezzati-Jivan et Dagenais (2013). Cette étude met l'emphase sur l'efficacité pour créer la structure de stockage de l'information de statistiques et sur la rapidité des requêtes dans cette structure.

2.4 Langages de définition

Dans cette dernière partie, nous allons voir l'état de l'art des langages de définition. Dans un premier temps, nous verrons les langages utilisés par d'autres applications de traçage et de sécurité. Ensuite, nous parlerons du langage XML et de sa capacité à s'intégrer dans l'environnement de développement Éclipse.

Un état de l'art des langages de scénario pour les systèmes de détection d'intrusion (IDS) et les traceurs noyaux a été fait par Matni (2009) et Waly (2011). Ils ont identifié six types de langages permettant de représenter des scénarios de sécurité : les langages déclaratifs, les langages impératifs, les langages basés sur les automates, les langages à logique temporelle, les langages par règles et les systèmes experts. Comme nous souhaitons surtout mettre l'emphase sur les langages permettant de modéliser un gestionnaire d'état, nous nous concentrerons ici sur les trois premiers.

7. Les commandes fondamentales de Linux - wiki.linux-france.org

2.4.1 Langages déclaratifs

Les langages déclaratifs permettent de créer des applications en décrivant le *quoi*, c'est-à-dire le problème (voir Lloyd, 1994), contrairement aux langages impératifs, qui expriment le *comment*. Le langage HTML, par exemple, décrit le contenu d'une page, et non comment l'afficher (positionnement, couleurs...).

Il existe plusieurs logiciels qui utilisent ce type de langage pour les analyses.

Snort

Snort (voir Roesch *et al.*, 1999) est un IDS libre au niveau du réseau. Il a été conçu pour capturer et surveiller les paquets sur le réseau. Ce logiciel utilise une syntaxe permettant de définir des règles pour détecter les intrusions. Dans ce contexte, l'utilisateur peut écrire de nouvelles règles qui utilisent différents champs des paquets réseau (adresse IP source, adresse IP de destination, les données du paquet, etc.). Ces règles sont divisées en deux parties, l'en-tête et les options :

1. L'en-tête permet de spécifier l'action qui doit avoir lieu lorsque la règle est satisfaite. Elle contient également l'IP source et l'IP de destination.
2. Les options permettent de choisir les champs d'application et fonctionnent comme des filtres sur les données.

SECnology

SECnology est un logiciel de sécurité permettant d'analyser des fichiers de journal (voir SECnology, 2014). Il est aussi possible de définir des règles avec son interface graphique. Un module spécifique, appelé SECalert, regarde ces règles pour détecter des comportements anormaux et effectuer des actions.

SECnology utilise également un langage déclaratif permettant de filtrer sur les champs possibles du journal.

On peut conclure néanmoins que ce type de langage seul est limité pour faire des analyses complexes à base d'états car tous les événements sont indépendants les uns des autres. Cependant, la simplicité de ce système peut être un bon compromis pour créer des fournisseurs d'état utilisables dans TMF.

2.4.2 Langages impératifs

Contrairement aux langages déclaratifs, les langages impératifs permettent de spécifier *comment* le but doit être poursuivi. Ce paradigme de programmation, plus expressif, est

utilisé par DTrace, SystemTap ou encore ASAX avec le langage RUSSEL.

RUSSEL

Le langage RUSSEL (RUle-baSed Sequence Evaluation Language) est utilisé par le projet ASAX [8] (voir Habra *et al.*, 1992). Comme son nom l'indique, ce langage est basé sur des règles prédéfinies. L'avantage de ce langage par rapport aux précédents est qu'il est possible d'appeler une seconde règle à partir de la première, ce qui offre plus de possibilités. Une limitation est tout de même présente pour ce langage : il n'est possible d'avoir qu'une seule règle active à la fois. Il est alors plus difficile de proposer des modèles avec plusieurs changements d'état indépendants les uns des autres.

DTrace

Nous avons vu dans la partie sur les traceurs que le logiciel DTrace est un traceur permettant de faire de l'instrumentation dynamique. Ce traceur utilise un langage de script, le *langage D*, pour définir les points de trace. Il permet de définir les conditions et les actions à effectuer pour chaque point de trace activé. Ces actions servent à spécifier les données collectées par le point de trace. La syntaxe du langage D est très proche du langage C et du langage de script awk (voir Balima, 2011) sauf que les conditions sont définies à l'aide de prédicats et non pas en utilisant des structures conditionnelles.

Les scripts D sont convertis dans un format binaire en utilisant un compilateur spécial avec un jeu d'instructions limité RISC conçu pour être facile à émuler. Le binaire est ensuite envoyé au noyau pour que DTrace le vérifie et active le point de trace.

Une utilisation de ce langage a été faite il y a plusieurs années pour créer des analyses spécifiques utilisant l'instrumentation dynamique (voir Luk *et al.*, 2005).

Le langage D est très spécifique au traçage, car il a été conçu pour ajouter des points de trace. Il ne permet pas vraiment de concevoir des analyses basées sur une machine à états.

SystemTap

SystemTap offre également un langage de script très semblable au langage D (voir Prasad *et al.*, 2005). Le langage utilisé possède une syntaxe très proche du C. Il supporte toutes les opérations de l'ANSI C.

La déclaration d'un point de trace se fait en deux parties. La première sert à identifier le point de trace. La seconde permet d'énoncer le code à exécuter lorsque le point de trace est rencontré. Les scripts de SystemTap sont ensuite convertis en C pour être compilés sous

8. Advanced Security Audit-trail Analysis

la forme d'un module noyau. Ce module est alors inséré dans le noyau et communique avec SystemTap pour recueillir la trace. On note qu'il est également possible de dépasser les limites du langage de script de SystemTap avec le mode permettant d'insérer directement du code C dans les scripts.

Comme pour DTrace, il permet surtout de spécifier comment sont placés les points de trace et n'est pas conçu pour construire des analyses complexes. Avec TMF, l'analyse de la trace est effectuée a posteriori. Il n'y a donc pas d'impact sur le traçage est les analyses peuvent être plus complexes.

2.4.3 Langages basés sur les automates

Nous allons maintenant explorer les langages basés sur les automates finis. Ces langages décrivent un automate fini pour résoudre les problèmes à l'aide d'un nombre fini d'états, de transitions et d'actions.

STATL

STATL, pour State Transition Analysis Technique Language, est un langage extensible permettant de décrire des patrons d'attaques dans le cadre d'un IDS (voir Eckmann *et al.*, 2002). Il fonctionne en définissant des *scénarios* qui contiennent des états et des transitions permettant de passer d'un état à l'autre. Ce langage est très adapté pour construire des analyses complexes en fournissant un niveau d'abstraction supplémentaire.

State Machine Compiler

Le State Machine Compiler (SMC) est une librairie disponible dans 15 langages de programmation (notamment C, C++, C# Java, Python) fournissant la possibilité de construire des machines à états (voir Rapp, 2014). Le langage est très similaire au langage STATL et offre une facilité d'intégration par un large choix de langages.

Ces langages se rapprochent beaucoup de notre besoin pour construire des analyses basées sur un modèle à états. Cependant, comme nous possédons déjà une machine à états dans notre projet qui est semblable à une base de données, ces langages ne sont pas très adaptés dans notre contexte. En effet, ces derniers mettent davantage l'accent sur l'état alors que nous souhaitons donner la priorité à l'attribut qui correspond à la propriété du système étudié.

2.4.4 XML

Le Extensible Markup Language (XML), soit le langage de balisage extensible, est un langage de balisage générique (voir Bray *et al.*, 1997). Ce langage est dit extensible car il permet de définir différents espaces de noms, c'est-à-dire des langages dédiés ou *Domain specific language*. L'objectif initial du XML est de remplacer son prédécesseur, le Standard Generalized Markup Language (SGML), en définissant un langage aussi générique mais plus simple d'utilisation. Le XML permet de faciliter l'échange automatisé de contenus complexes entre des systèmes d'information différents. L'intérêt du XML est de normaliser l'utilisation des balises, ce qui a permis de créer des analyseurs syntaxiques standards.

Schémas de définition

Afin de faciliter la définition de nouveaux langages à partir du langage XML, plusieurs langages de schéma ont vu le jour. Ces langages permettent de définir les règles de validation d'un langage dédié utilisant le XML.

Le premier langage de définition pour le XML est le Document Type Definition (DTD). Ce langage est issu du SGML. Le DTD permet de définir un document à partir d'une syntaxe texte. Cependant, ce langage possède une limitation importante pour la définition d'un langage dédié complexe : il ne permet pas de gérer la documentation associée au langage.

Pour cette raison, le langage de schéma XSD, pour XML Schema Definition, a vu le jour (voir Thompson, 2004). Ce successeur, qui repose également sur le langage XML, permet de définir complètement un langage dédié. De nombreux outils sont disponibles pour valider des fichiers XML à partir d'un fichier XSD.

2.4.5 Utilisation du XML dans Éclipse

Le travail de ce mémoire est destiné à être intégré dans TMF. Or ce visualiseur est développé comme un greffon de Éclipse, il est donc tout naturel de s'intéresser à l'utilisation du langage XML dans cet environnement de développement.

Extensibilité

À l'origine en 2001, le but du projet Éclipse est de proposer un cadre d'application pour créer des environnements de développement. En 2004, suite à la pression de la communauté, il a été décidé d'étendre le cadre d'application à la création de n'importe quelle application client avec *Eclipse Rich Client Platform* (voir Eclipse Foundation, 2014).

Cette évolution a permis de structurer la façon d'ajouter des greffons dans Éclipse en introduisant un mécanisme générique d'extensibilité : tout greffon peut se déclarer extensible

et tout greffon peut étendre un autre greffon extensible.

Afin de permettre ce mécanisme, le XML est très présent dans la définition des greffons de la plateforme Éclipse (voir Bolour, 2003). En effet, le composant de base pour ajouter un nouveau greffon est le manifeste décrit dans le fichier `plugin.xml`.

Grâce à ce fichier, un greffon peut devenir extensible en déclarant un point d'extension. Pour réaliser cette opération, il doit notamment indiquer un schéma XSD qui définit la grammaire utilisée pour le point d'extension. Ensuite, les greffons qui souhaitent utiliser ce point d'extension déclarent une extension dans le fichier `plugin.xml` en respectant la syntaxe associée.

Les extensions peuvent concerner diverses fonctionnalités du logiciel, telles que les menus, les vues, les fenêtres interactives, ou encore des propriétés (types de traces dans TMF).

Une interface entre deux outils

Par ailleurs, le langage Java fournit un ensemble de classes de base permettant d'importer, d'exporter et de manipuler des fichiers XML. Ainsi, le XML est un bon outil pour faire l'interface entre deux systèmes hétérogènes.

Un exemple d'utilisation dans TMF est un formulaire pour créer des filtres sur les événements (voir Figure 2.6). Il est possible définir interactivement un filtre qui sera sauvegardé et échangé en XML avec le moteur de recherche pour appliquer le filtre.

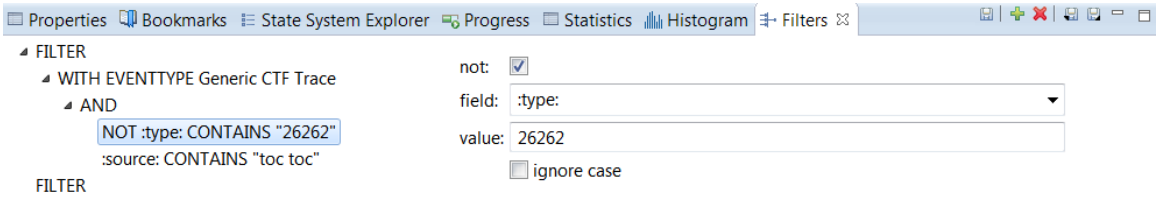

Figure 2.6 Vue dans Éclipse permettant de créer un arbre avec des champs prédéfinis qui sera stocké en XML

La capacité de l'XML est être un langage permettant de faire le lien entre deux parties du logiciel sera largement utilisée dans notre étude.

CHAPITRE 3

MÉTHODOLOGIE

La création d'un nouveau langage dédié soulève différents enjeux qui doivent être pris en compte dans la méthodologie pour réaliser notre étude. Il est nécessaire d'analyser les conditions de travail actuelles pour identifier les points qui peuvent être améliorés.

Dans cette partie, nous présenterons les critères utilisés pour réaliser notre étude. Nous définirons d'abord l'expressivité d'un langage, puis son utilisabilité. Enfin, nous regarderons quels sont les critères de performance qui s'appliquent dans notre étude présentée dans le chapitre 4.

3.1 Expressivité

L'expressivité du langage permet de déterminer la capacité de la grammaire définie à répondre à la problématique visée par le langage.

En informatique, la capacité d'un langage de programmation à permettre de calculer n'importe quel algorithme a été définie formellement par Alan Turing et Alonzo Church en 1936 (voir Turing, 1936). Cette définition repose sur un modèle abstrait de calculateur, appelé machine de Turing. La thèse de Church postule qu'il est possible de résoudre tout problème défini par un algorithme à l'aide d'une machine de Turing.

Depuis, afin de caractériser l'expressivité d'un langage de programmation, il est possible de montrer que le langage est complet au sens de Turing, c'est-à-dire que le système formel possède au moins une puissance de calcul équivalente à une machine de Turing. On notera cependant que la machine de Turing suppose l'usage d'un *ruban* de mémoire infini. Le fait que les ordinateurs possèdent une quantité de mémoire finie est une limitation généralement admise.

La plupart des langages de programmation sont complets au sens de Turing, comme le C, C++, Java ou encore le Pascal. Il existe cependant des langages qui ne sont pas complets tel que les langages reconnaissables, comme les expressions régulières. Les langages dédiés peuvent ne pas être complets par choix afin d'être déterministes (absence de boucle infinie), ce qui sera le cas pour notre langage.

3.2 Utilisabilité

L'utilisabilité est définie par la norme ISO 9241-11 [1] comme :

> "le degré selon lequel un produit peut être utilisé, par des utilisateurs identifiés, pour atteindre des buts définis avec efficacité, efficience et satisfaction, dans un contexte d'utilisation spécifié."

Dans notre contexte, l'efficacité est directement mesurée avec l'expressivité du langage, c'est-à-dire si l'utilisateur est capable de traiter son cas d'utilisation avec notre nouvel outil. Nous allons montrer dans notre étude les possibilités et les limites du langage pour caractériser cette efficacité.

L'efficience est un critère plus subjectif à déterminer. Il mesure la capacité de l'utilisateur à réaliser son objectif avec un effort minimal. Pour le travail réalisé dans ce mémoire, l'objectif semble atteint. Nous proposons une solution qui ne nécessite plus l'effort de développer de nouvelles classes Java pour créer de nouvelles analyses. L'utilisateur devient capable de créer, d'utiliser et d'échanger ses analyses directement à partir de l'interface graphique du logiciel et n'a plus besoin de l'installer en mode développeur. Cette solution permet de réduire significativement l'effort fait par l'utilisateur pour développer de nouvelles analyses.

La satisfaction correspond au confort subjectif de l'interaction avec l'utilisateur. Ce point reste encore à l'étude. La solution a été conçue pour définir un format d'échange des données. Ce format réalisé en XML a été fait avec le but de construire une interface capable de générer automatiquement les fichiers XML en respectant la grammaire introduite dans ce mémoire.

Pour le moment, la création et l'édition du fichier XML se fait directement à l'aide d'un éditeur. Le fichier de spécification XSD apporte la documentation et la validation nécessaire.

3.3 Performance

Le troisième critère important pour la réussite de ce projet est la performance. En effet, comme les données manipulées sont d'une taille très importante, il est nécessaire d'avoir des algorithmes performants.

Les outils développés pour ce mémoire s'intègrent dans TMF. Il a été important de mesurer les performances des outils existants afin de les comparer avec la solution proposée. L'objectif principal est de montrer que l'introduction du langage déclaratif ne diminue pas les performances de TMF.

1. `http://fr.wikipedia.org/wiki/Utilisabilité`

Cependant, comme nous avons également introduit des nouvelles formes d'utilisation du gestionnaire d'état, il a été nécessaire de développer de nouvelles formes de requêtes afin d'obtenir des performances meilleures dans toutes les situations. Ainsi pendant l'étude, des gains en performance de l'ordre de 30% sur le temps de création de l'arbre du gestionnaire d'état ont été faits ainsi qu'un gain de près de 20 fois dans la vitesse des requêtes courantes pour afficher la vue.

Ces gains de performance ont permis de construire un modèle toujours plus complexe, avec des tailles de trace 10 fois plus importantes, sans dégrader les performances du logiciel.

Nous allons maintenant détailler dans le chapitre suivant le langage que nous proposons pour notre étude. Ce chapitre est constitué d'un article soumis à la revue *Advances in Software Engineering* de Hindawi. Il montre les possibilités de construire un modèle à base d'état à partir d'une trace système en utilisant un langage déclaratif.

CHAPITRE 4

ARTICLE 1 : DECLARATIVE SPECIFICATION FOR CONSTRUCTING THE MODELED STATE FROM A SYSTEM EXECUTION TRACE

Authors

Florian Wininger
Ecole Polytechnique de Montréal
florian.wininger@polymtl.ca

Michel Dagenais
Ecole Polytechnique de Montréal
michel.dagenais@polymtl.ca

Submitted to Advances in Software Engineering (Hindawi).

Keywords Performance analysis, Tracing, System state analysis, Declarative language.

4.1 Abstract

With newer complex multi-core systems, it is more important than ever for a developer to understand his application. However, with multiple abstraction levels, it becomes increasingly difficult to find the exact location of performance or security problems. Tracing tools provide generic analysis views to help understand these problems. The developer is the one who best knows his application. We therefore propose in this paper a declarative specification and optimized analysis tool architecture to create new custom analyses. This enhanced framework builds custom analyses based on a specified modeled state, extracted from a system execution trace and stored in a special purpose database. This proposed solution can be used to add custom state information in a new state model, display this model as many alternate representations (Gantt chart, statistics, etc.), and filter this data to detect some patterns. Several sample applications are shown, with different operating systems, as well as combining kernel and user-space events.

4.2 Introduction

Modern tracing tools extract useful data about the runtime behavior of a system or an application. The concept of tracing is to insert trace points or probes at specific locations in the source code. Those trace points send information when encountered during program execution. The LTTng tracer (Linux Tracing Toolkit Next Generation) Desnoyers et Dagenais (2006) Desnoyers et Dagenais (2008) was developed by the DORSAL lab at Ecole Polytechnique de Montreal. This tracer, available for the Linux Kernel, is optimized for low overhead and collects kernel and user-space events.

When systems with multiple cores and several computer nodes are traced, the collected data is very large. In this context, it is very important to have efficient analysis and visualization tools to extract the useful information to solve a problem. There are several tools available to give a graphical representation of different runtime metrics.

In the proposed architecture, we use a state-based approach to model the system and give the user a comprehensive image of the application state during the execution time. For example, the state of a process may change over time between states *start*, *running*, *waiting* and *stopped*. This changing state of a process is stored in an "attribute" that will be named *status*. The approach to index and retrieve the system state history has been used previously Cohen *et al.* (2005) Cohen *et al.* (2004). The states selected to model the system are very important, and depend on the system and the problem we want to investigate. To study a performance degradation for example, we should track states of important resources (e.g. what are the CPU usage, the currently running thread, the files being accessed, or the network usage). Such metrics can help administrators understand the problem and possibly find a way to eliminate the underlying cause.

Each system is unique, and existing analysis views only cover the most typical contexts. Nevertheless, as problems are often complex, it is possible that these analyses do not help targeting them sufficiently. We propose in this paper a tool architecture to allow the developer to easily extend the modeled state according to the application's characteristics. This custom state is used to better display the information, and to directly find the position of the problematic events in a trace by filtering.

The remainder of the paper is organized as follows : after reviewing related work and the existing infrastructure, we present the specification of the proposed declarative language and detail the sample implementation. Then, we discuss several possible analyses and visualizations using the enhanced state, and we validate the flexibility and performance of this solution. Finally, we conclude and outline possible future work.

4.3 Previous Work

An approach to model the state from a system trace has already been studied Montplaisir *et al.* (2013) and implemented in the Eclipse Tracing and Monitoring Framework, TMF [1]. It is based on a state manager and a special purpose database, used to efficiently store, navigate and display the state in the analysis software.

4.3.1 State System

The general idea of the State System architecture is to incrementally extract the information of each relevant trace event and create different state values. The information flow is described in Figure 4.1.

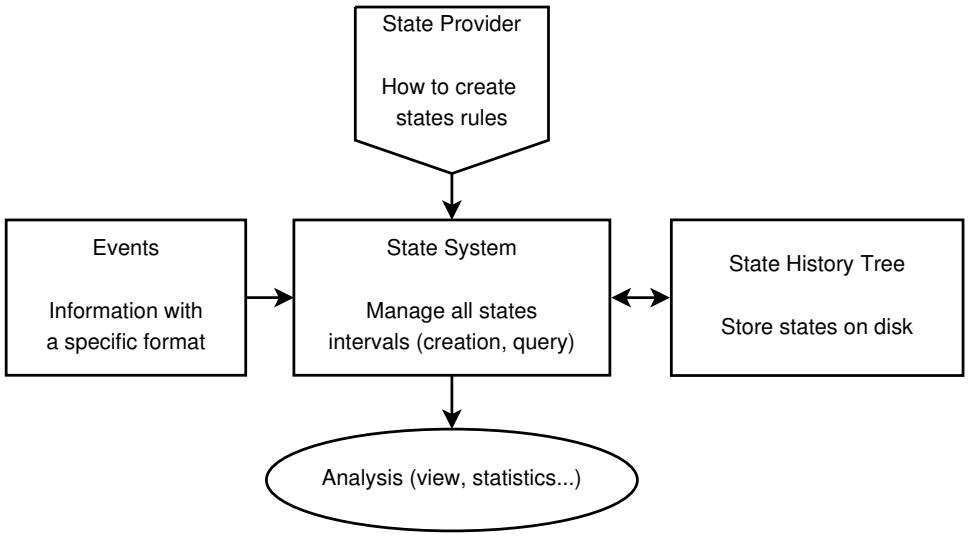

Figure 4.1 Architecture of the State System

Attribute Tree

We define an *Attribute* as a generic term to represent the smallest part of the state model. An attribute can be any system property, for example "the thread status", "which thread is on CPU1", "if a file is open" or "the stack of a thread". It depends on the studied system.

The attributes are organized as a tree called the *attribute tree* in order to be able to manage a large number of attributes. These represent together the complete system state. This *attribute tree* is just a level of abstraction to access attributes, each containing a specific system property. An example of an *attribute tree* is shown in Figure 4.2.

1. `http://www.eclipse.org/linuxtools/projectPages/lttng/`

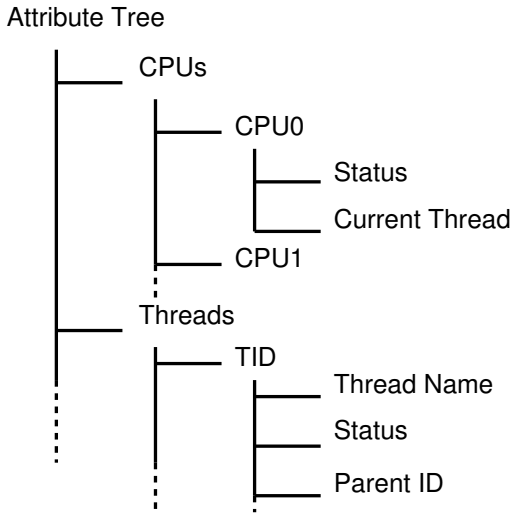

Figure 4.2 Example of an attribute tree

In this data structure, all attributes are accessible through a specific path, like in a file system, (for example "`/CPUs/CPU0/ Current Thread`"). This allows the analysis to make queries to access an attribute.

State Provider

The key element of the modeling process is the *State Provider*. This element produces *state changes*, based on events received, to create new state values. Each attribute value, between two changes, represents a *state interval*. For instance, for a "linux_sched_switch" event, we update the currently running thread on the associated CPU. This state change describes the end of the previous state interval, with the scheduled-out thread, and the beginning of a new state interval, with the scheduled-in thread.

In the existing implementation, custom java code was written specifically to define how states changed based on a Linux kernel trace Montplaisir *et al.* (2013). Between specific Linux kernel versions, this code sometimes needed to be adjusted because the kernel code and associated trace points had changed. Different applications (e.g. other operating system kernels, Database systems, Web servers) would need their own custom *State Provider* Java code. Thus, the major disadvantage of this method is the necessity to develop new classes when you want to add new analyses.

4.3.2 State History Tree

With the huge trace data size, a special purpose database was designed to store all produced state intervals on hard disk Montplaisir-Goncalves *et al.* (2013). This *State History*

Tree allows the state system to make fast queries for any attribute at any time during the trace. Furthermore, all state intervals are inserted by sorted end time. The State History Tree uses this property to optimize its layout for fast access on a rotational disk. This property obviates the need for re-balancing the tree, but preserves the property of logarithmic search. As a result, this data structure is well optimized to be used with trace files as large as 1TB.

4.3.3 Visualization

Several tools exist to visualize and analyze such traces. Viewers like LTTV (Linux Tracing Toolkit Viewer)Deschênes *et al.* (2008), Jumpshot Zaki *et al.* (1999) or Triva Schnorr *et al.* (2009) display different runtime metrics (CPU usage, memory consumption, critical path analysis, etc.).

Similarly, it is possible to view the data stored in the state system. Several views are available to present data : statistics, Gantt charts or histograms. You can see in Figure 4.3 the data representation for Linux kernel traces : CPU usage, threads activities, statistics for the number of events, etc.

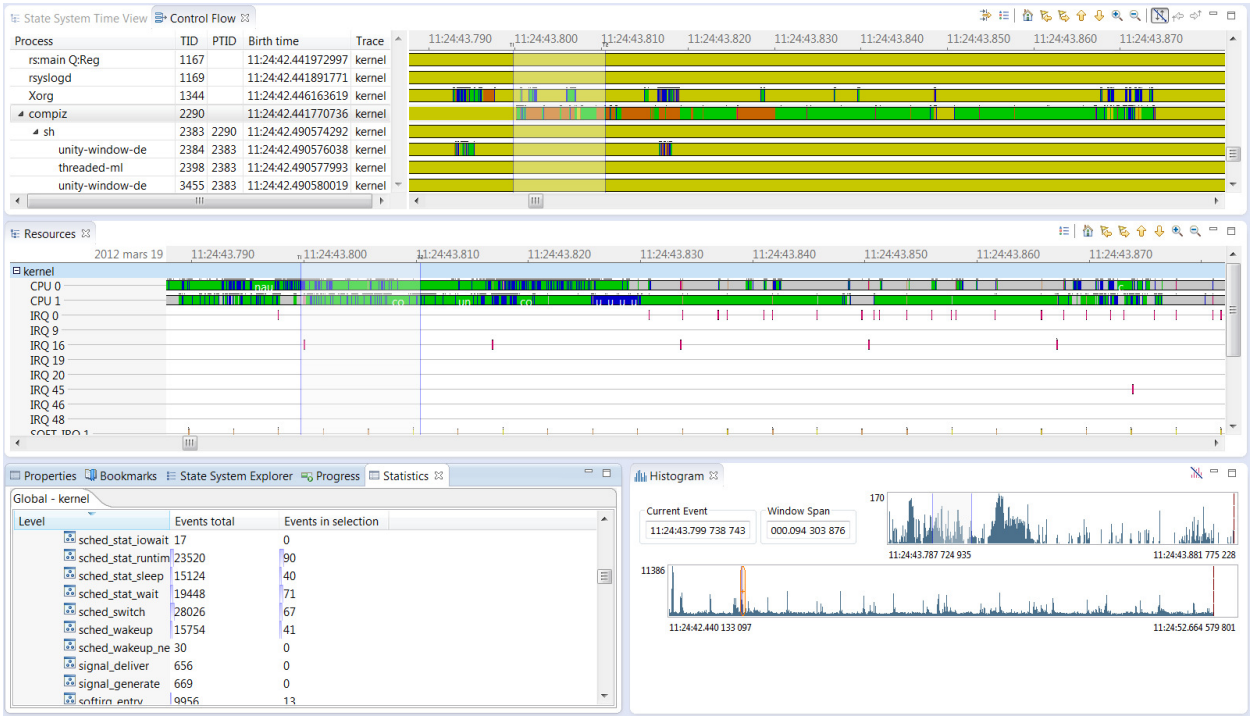

Figure 4.3 Multiple data views for a Linux Kernel Trace display with TMF

However, a limitation of these tools is that it is only available for a particular trace type generated by a specific tracer. In the new proposed architecture, the state provider, state

tree, state history tree database and state display views are completely generic, and thus easily customizable.

4.3.4 Description languages

There are many types of languages dedicated to system analysis. Interesting reviews of trace analysis systems are available from Matni Matni et Dagenais (2009) and Waly Waly (2011).

Declarative languages for patterns in network traces and logs are used by SNORT Roesch *et al.* (1999) or SECnology SECnology (2014). SNORT is an open-source Network Intrusion Detection System based on a collection of rules. This software provides a simple declarative syntax for defining intrusions in network connection packet traces. Nonetheless, by looking at each packet in isolation, this technique alone is not very efficient for defining complex analysis such as those in State Providers.

Imperative languages like RUSSEL (RUle-baSed Sequence Evaluation Language) Habra *et al.* (1992) offer better expressiveness. Rules can trigger other rules. If rules are viewed as procedures, it is similar to procedural languages. Another language is the D language, designed by DTrace Cantrill *et al.* (2004) to dynamically define the instrumentation probes. However, this is more of a generic imperative language. SystemTap Eigler et Red Hat (2006), another Linux kernel tracer, also provides a similar imperative scripting language, triggered by kernel-level events.

Automata-based languages are closer to the requirements of defining state transitions from events. This kind of language uses a finite state machine to describe the problem, with states, transitions and actions. STATL (State Transition Analysis Technique Language) Eckmann *et al.* (2002) is a good example of a generic state machine diagram language that is extensible and usable by different applications. However, these languages are not necessarily adapted to use with a backend like a State System, which is closer to a database. On the other hand, query languages like SQL are limited to tabular data.

In this context, we have defined a domain-specific language to convert events into states.

4.4 Specification of the descriptive language

The main idea of this contribution is to define a flexible language to model the state from a system execution trace. This new descriptive language is able to create states used by the state system, and provides new analyses in a specific context. For this, we introduce some basic operations to modify the state attribute tree. These operations allow accessing an attribute in the attribute tree by using a path described by constants, event data and values

which are already present in the state tree. Then, we use the event data and the current state system to assign a new value to an attribute. This assignment can be conditional on the event data or on current values in the state system. Furthermore, the implementation of these new features does not decrease the performance of the state system.

4.4.1 Convert an event into states

As mentioned earlier, we use event fields to create new states. This mechanism called *state change* is the main element of our language. A state has a beginning and continues to exist until its end. In our particular case, the end of a state assigned to an attribute corresponds to the beginning of the next state. Thus the *state change* is a kind of temporal border between the beginning of the new state and the end of the previous one. The time of this border is the time of the event associated with this state change.

A simple case of an *event to state* transcription is when a boolean state, with associated events, corresponds to the entry and exit of the state. For example, for system calls, we have an entry event that changes the state into "`running in kernel mode`", eventually followed by an exit event that changes the state into "`running in user mode`".

This case is important because it is the easiest way to create a state with beginning and end. We can think of a custom user-space trace where a user would add trace points for the start and end of each function. It becomes very easy to know at any moment the current function executed by the program, or even the complete call stack.

In the most common case, the state changes can be seen as transitions in a finite state machine. Therefore, the conversion of an event into a state is equivalent to a conditional assignment in the state system.

4.4.2 Language definition

We previously defined the attribute tree of the state system, and the path associated with each attribute. From there, we can define the operations necessary to model the system.

Access to attribute values

In order to access an attribute, we use a path such as the following :

```
/Threads/100/Status
```
(4.1)

To make a query and replace the expression by the result, we use `${}` as in (4.2). This queries another attribute, for example to be used itself as a path component. Here thread

`100` is the current thread on CPU 1.

```
/Threads/${/CPUs/1/CurrentThread}/Status
```
(4.2)

As mentioned above, when the state system is built, we use the event fields to extract the information to put in the state system. Therefore, we need the possibility to access an event field to make a query in the path, as in (4.3). In this syntax `event/*` is used to access the event fields, and not the `/event/` path in the attribute tree.

```
/Threads/$/CPUs/${event/cpu_id}/CurrentThread/Status
```
(4.3)

In practice, for kernel traces, some information such as the thread id is not available in all events. It is thus necessary to use the context switch events to extract this information and store it in the state system for each CPU core. It is then possible to have the current thread id for each event by simply knowing its CPU's number. Thus, as in (4.3), it is possible to get the current thread status through a query in the state system, accessing an event field.

Assignment

Another possible operation is the assignment of an attribute. This operation changes the value of an attribute, ending the previous state interval and starting a new one with the new value.

```
/CPUs/${event/cpu_id}/Status = RUN_IN_USERMODE
```
(4.4)

The value can be a constant, as in (4.4), another path in the state system, or an event field, as in (4.5).

```
/Threads/${event/tid}/Exec_name = /event/exec_name
```
(4.5)

Condition

The last element is for providing conditional state changes. The simplest condition is the type of the event. It is necessary to sort the different state changes by event type, to allow the user to easily correlate the changes with a trace point in the source code, when reviewing the state provider declarations.

Within a specific event type, state changes can also be conditional, at a second level, based on state values or event fields. The same syntax to access the variables is used, with classical boolean operators `AND`, `OR`, and `NOT` for conditions.

```
/File/${event/fd}/Status == OPEN AND /event/filename == ".passwd"
```

It then becomes possible to choose the conditions for applying state changes, by using the information contained in the event and the information already present in the state system.

4.4.3 Language limitation

This language allows some operations to access and assign the memory of the state system. It can also make conditions. However, we did not include unrestricted conditional or unconditional branching. This prevents looping and accordingly infinite loops, insuring that the processing time is finite. This only allows a finite number of state changes for each event. Because of this limitation, our descriptive language is not Turing complete.

4.5 XML definition

We have shown in the previous section that it is possible to declaratively define the process of converting an event into states. We will now see that it is possible to use this syntax to completely define a model for the behavior of an operating system or an application. In addition, we will expand this functionality with filtering and data visualization.

Therefore, to facilitate future functionality extensions, it was decided to use XML with XSD [2] schema definition for the exchange of data between the state system and the user. A user interface could provide an intuitive way to create new models and would be able to generate the needed XML format.

4.5.1 State Provider

The state provider is the part that defines how to convert events to state changes stored in the state system. However, the state system is only an interface between the trace and the view, it is thus necessary to specify the trace type and how the analysis is used to display information. This information is added in a header section of the state provider.

State values and Locations

The first step to create a new state system is to choose the state values which correctly describe the modeled system. Values can be abstract states like `OPEN, CLOSED, RUNNING, STOPPED`, or a string that contains a payload like the executable filename for a process. For space optimization, abstract values are stored as integer values. Two examples of state values follow :

```
<stateValue name="RUNNING" value="1" />
```

2. In the package org.eclipse.linuxtools.tmf.analysis.xml.core

```
<stateValue name="STOPPED" value="2" />
```

When these state values are used, the state provider stores the integer 1 or 2 as value for the state interval. This system is similar to a map with a key and a value.

The second step is to define the organization of the attribute tree. In most cases, the user wants to store a list of indexed properties, like the status of all CPUs and threads, or the list of opened files. This vision is certainly reduced as compared to the expressiveness of the declarative language, but it is a common use and a good starting point. It enables an easy overview of the data, using a table of statistics or a Gantt chart.

In this context, we use a path with a wildcard, like `/Thread/*/Status`, where each possible value represented by `*` is a unique index, here the thread id. In this case, the index is obtained from an event field :

```
/Threads/${event/tid}/Status
```

The corresponding XML syntax is :

```
<location id="CurrentThreadStatus">
   <attribute constant="Threads" />
   <attribute eventfield="tid" />
   <attribute constant="Status" />
</location>
```

Frequently occurring complex value extraction expressions can be associated with a shortcut name. Although not mandatory, shortcuts may be used in state change declarations for conciseness and clarity purposes.

Event Handler

While the event type could have been yet another field subject to conditions, it was decided to have an explicit *event handler*, a top-level structure that defines a namespace for an event type. This choice structures and simplifies the addition of rules for a new trace points in the source code. It also helps to quickly check what types of events are needed for an analysis. Thus, the new user can know which events to activate when collecting a new system trace.

In our XML syntax, the Event Handler is a container for state changes :

```
<eventHandler eventname="sched_switch">
  <stateChange>
    <attribute location="CurThreadStatus"/>
    <value int="$RUNNING" />
  </stateChange>
  <stateChange>
    <attribute location="PrevThreadStatus"/>
    <value int="$STOPPED" />
  </stateChange>
</eventHandler>
```

In this example, the *sched_switch* event contains two changes. The first one updates the status of the current thread to "running" and the second one stops the previous thread.

State Change

The last part of the state provider is the transcription of the state changes defined above. This construction contains a path and a value, possibly with a condition.

For example :

```
/Threads/${event/tid}/exec_name = /event/execname
```

```
<stateChange>
   <attribute constant="Threads" />
   <attribute eventfield="tid" />
   <attribute constant="exec_name" />
   <value eventfield="execname" />
</stateChange>
```

A condition can be added, as in the complete example of the next section.

Example

Here is a simple example with user-space LTTng-UST instrumentation Blunck *et al.* (2009). The objective is to debug an application to know when it works. We add two trace points : one at the beginning called `application:start`, and one at the end called `application:end`.

Then we define two states : `RUNNING` and `STOPPED`. We know that we want to launch several instances of the software and show them in a Gantt chart view, so we can use an `Application/*/Status` schema to store the data.

The source code becomes :

```
<stateprovider analysisid="app.ust">
   <head>
      <tracetype id="ust.ctf" />
      <view id="gant.view" />
   </head>

   <stateValue name="RUNNING" value="1" />
   <stateValue name="STOPPED" value="0" />

   <location id="App_Status">
      <attribute constant="application" />
      <attribute eventfield="pid" />
      <attribute constant="Status" />
   </location>

   <eventHandler eventname="app:start">
      <stateChange>
         <attribute location="App_Status" />
         <value int="$RUNNING" />
      </stateChange>
   </eventHandler>
   <eventHandler eventname="app:end">
      <stateChange>
         <attribute location="App_Status" />
         <value int="$STOPPED" />
      </stateChange>
    </eventHandler>
</stateprovider>
```

We show the result in the TMF view in Figure 4.4.
You can see in green the active processes and in grey the stopped ones.

4.5.2 Filtering

We have added an effective mechanism to navigate the computed system state. It was indeed interesting to add filtering functions, for example in order to specify triggers to help

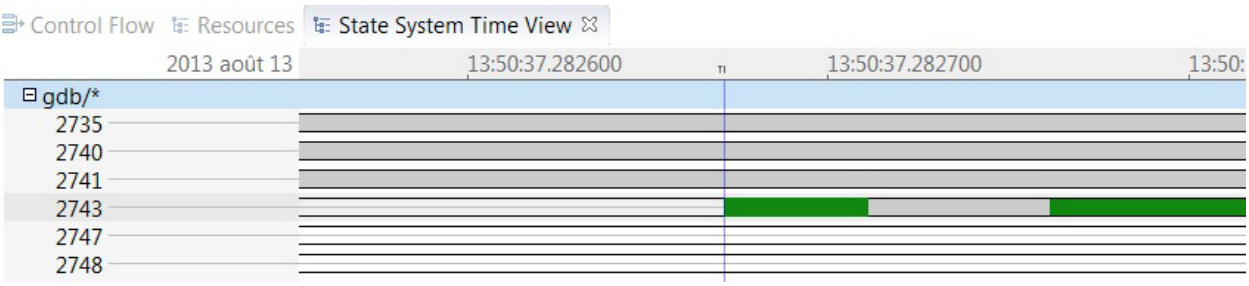

Figure 4.4 Example of a UST instrumentation

the user to debug his application, or to detect security attacks.

This detection pattern is also a kind of finite state machine which uses the data in the state system. This is why we can express these filters through the same syntax.

These filters create new virtual states which help to explain the state intervals defined by the state provider. They are stored in an in-memory state system backend with a specific folder `/Filters/`. We don't use a persistent storage, so the filter must be recalculated at every reload of the viewer.

The following example shows a filter able to find when a specific application is preempted by a lack of CPU resources. The new virtual state `BLOCKED` can be used to highlight the interesting portion of the trace.

```
<filter name="filter_1">
   <if>
      <attribute location="App_Thread" />
      <attribute constant="Status" />
      <value int="$STATUS_WAIT_FOR_CPU" />
   </if>
   <then>
      <attribute location="Filter" />
      <attribute constant="Blocked" />
      <value int="$BLOCKED">
   </then>
   <else>
      <attribute location="Filter" />
      <attribute constant="Blocked" />
      <value int="$UNBLOCKED" />
   </else>
```

```
</filter>
```

Because the virtual states have the same characteristics as state intervals, we can use the same views to display them.

Moreover we reuse the state intervals already stored in the history tree to quickly produce the results for a time range. In addition, this filtering can be used to add bookmarks in the trace, helping the user to navigate directly where a problem is detected.

4.5.3 Views

The state history tree is used to store temporal data. Therefore, it is easy to use this data for Gantt charts, representing resource states (here attributes) as a function of time. The Gantt chart view is currently used to visualize the CPUs or Threads activities Montplaisir *et al.* (2013). In the new proposed architecture, with the declarative state provider, we have replaced this view with a generic version to display lines defined by attributes with the pattern `folder/*/display_attribute`. In addition, we have added optional settings in the XML header to specify parameters for each state like coloring, tooltips, etc.

A second way to use the state system and add information for the user is to provide statistics. A complete study of the approach has already been presented Ezzati-Jivan et Dagenais (2013). Several useful metrics are easily extracted from the state tree. The most obvious are the time spent in a state and the relative frequency of each state. For example, it is possible to define states, or virtual states with filters, to measure the time spent by a process in state *"wait for a CPU"*. Thereby we have developed a view to display XY-charts (plot, bar chart or histogram). For this type of chart, we always use the time as X-axis. Then, we use the integer data contained in a specified attribute to define the associated Y values for the curve. We can then use the state system to display the memory consumption, or any cumulative statistic that can be defined by states or events.

4.6 Applications and performance analyses

The new proposed architecture, with the declarative language and generic use of the state system and views, was implemented in TMF. In this section, we will examine applications that have been achieved with this new tool. The tests to validate the performance of the implemented tools have been performed under Ubuntu Linux 12.04, on a dual quad-core Intel® Xeon® E5405 @2Ghz with 8GiB of RAM.

4.6.1 Generic Kernel Model

The first validated success of this new tool is the generality of the syntax. We were able to easily represent the Linux Kernel model with our XML syntax. Thus we were able to compare the conciseness and performance between the XML and hardcoded Java versions. Since the Linux Kernel model was our starting point, it was expected that the required expressiveness would be provided. More impressive was the fact that the new declarative language was used to easily interface Event Tracing for Windows (ETW) kernel traces to our new TMF architecture. We can now support Windows operating system kernel traces with the same level of information.

Construction Time of a history tree

In this benchmark, we propose to evaluate whether there is a performance degradation between a state provider implemented using Java and a state provider using the proposed XML syntax.

For this, we used two kernel traces : a 13.4 MiB trace available as a CTF sample in LTTng website [3] and a 100 MiB kernel trace. The tests were repeated 25 times to get an average value and standard deviation.

Tableau 4.1 Construction Time of the history tree for a 13.4 MiB kernel trace.

Trace 13.4 MiB	Java	XML
Average time (s)	8.687	8.979
Standard Deviation (s)	0.218	0.277
Min (s)	8.263	8.387
Max (s)	9.141	9.797

Tableau 4.2 Construction Time of the history tree for a 100 MiB kernel trace.

Trace 100 MiB	Java	XML
Average time (s)	49.359	50.025
Standard Deviation (s)	1.034	1.140
Min (s)	47.054	44.325
Max (s)	52.670	52.427

The results in Table 4.1 and 4.2 show that the XML version is slightly slower. However, the difference is smaller than the standard deviation between the different tests. Variations

3. `http://lttng.org/download`

between instances are mainly associated with the garbage collection of the necessary objects to create and store state intervals.

Linux and Windows comparison

The way to represent an operating system with a Gantt chart view, and to describe threads activities, is fairly common. However the strength of our model is that it can be easily interfaced to all platforms with a kernel tracer. By studying the ETW tracer on Microsoft Windows, we have shown that this tracer has equivalent events and can be used to model the system in the same way. It was then possible, with a simple revision of the XML file, to get the same views, already available for Linux, with Microsoft Windows.

This shows the independence of the OS, and identifies the specific group of events needed to define new analyses.

To compare the behavior of the two operating systems, we designed a simple test. Every second, we start a new process that makes a *CPU burn*. The objective is to see how these increasingly numerous threads are distributed on a computer with 4 CPUs. The result is shown in Figures 4.5 and 4.6.

4.6.2 Multi-level tracing

A second usage scenario for our new language was to define more complex models based on the generic kernel model. For this objective, we have tried two different applications to integrate another source of events.

UST and Kernel

We chose an already instrumented application whose architecture behavior is only visible with both user-space and system traces. Indeed, it uses both numerous system-level threads and user-level task queues. This application, Google Chromium, is well documented Chromium Project (2014). The UST model of Chromium has already been defined and is used by their internal tracer [4]. This model use two event types to know the beginning and end of a code portion. Not all functions are instrumented, only the key functions of the application.

In our case, we want to use these events to maintain a stack of the functions running for each thread. We use the *Begin* event to push the function name on the stack, and the *End* event to pop this stack.

4. `chrome://tracing` in the Chromium browser

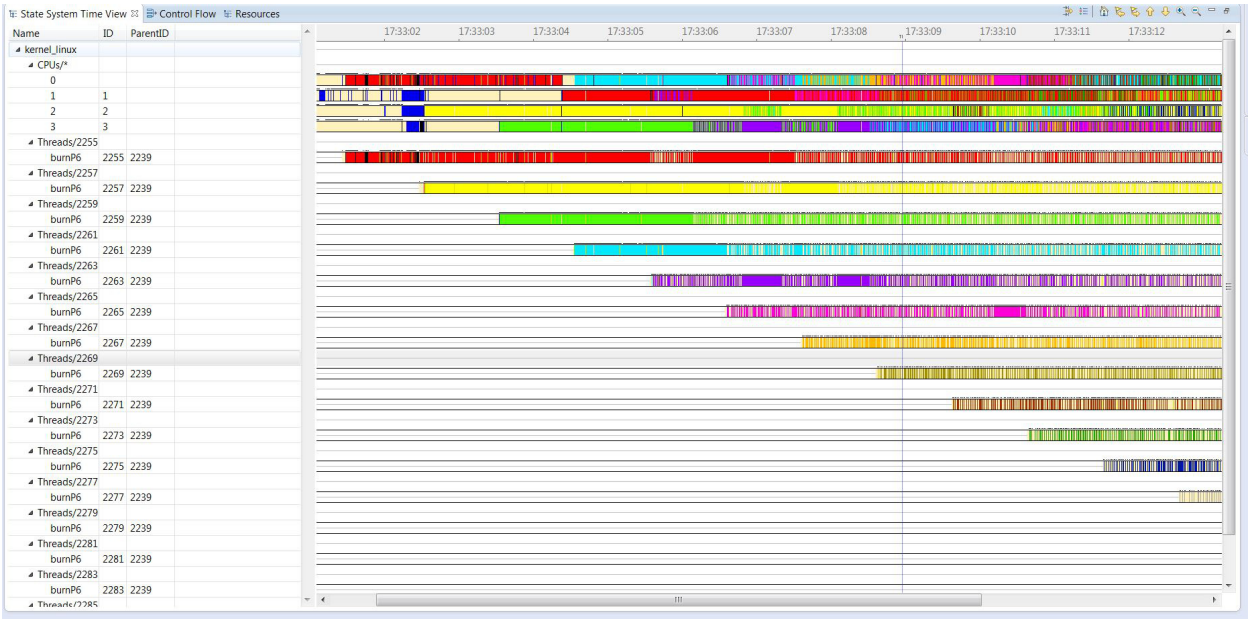

Figure 4.5 Thread activities view in Linux.

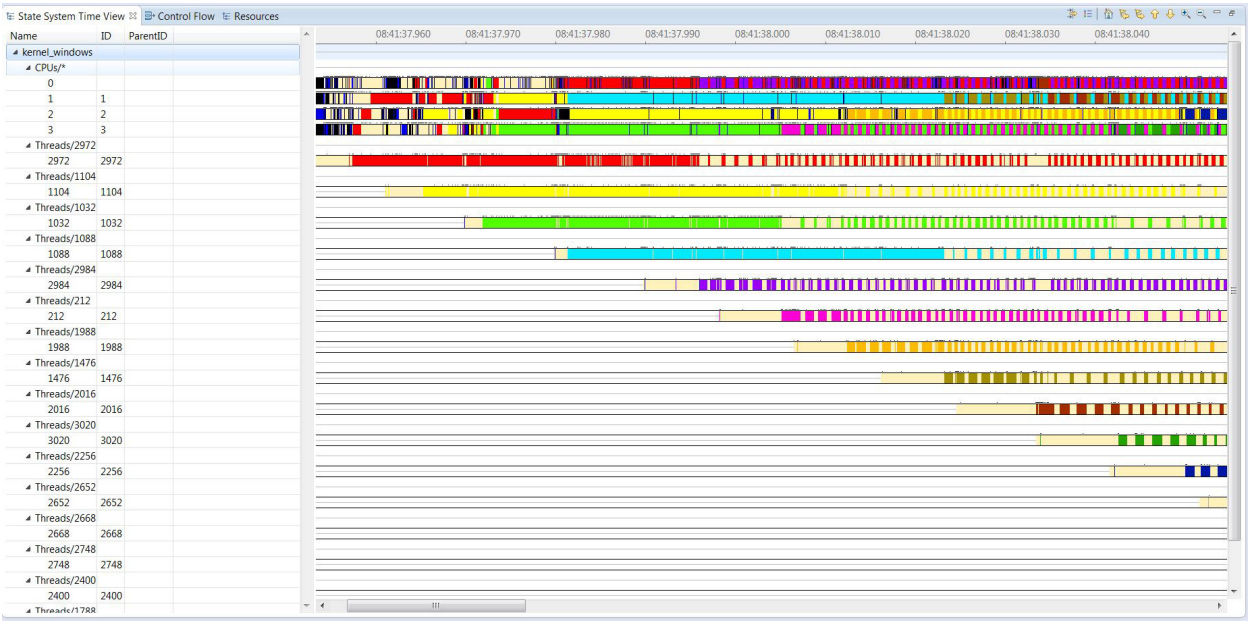

Figure 4.6 Thread activities view in Windows.

By combining the kernel and UST data, we know what functions are running in each thread, but we also know when threads are preempted or which system calls are executed for each function. In Figure 4.7, we see the current stack in the “State System Explorer” view. At this moment, we see that the stack depth is 4, and the top is *OnDispatchMessage*. The name of this function is also used as label in the green portion in the Gantt chart view.

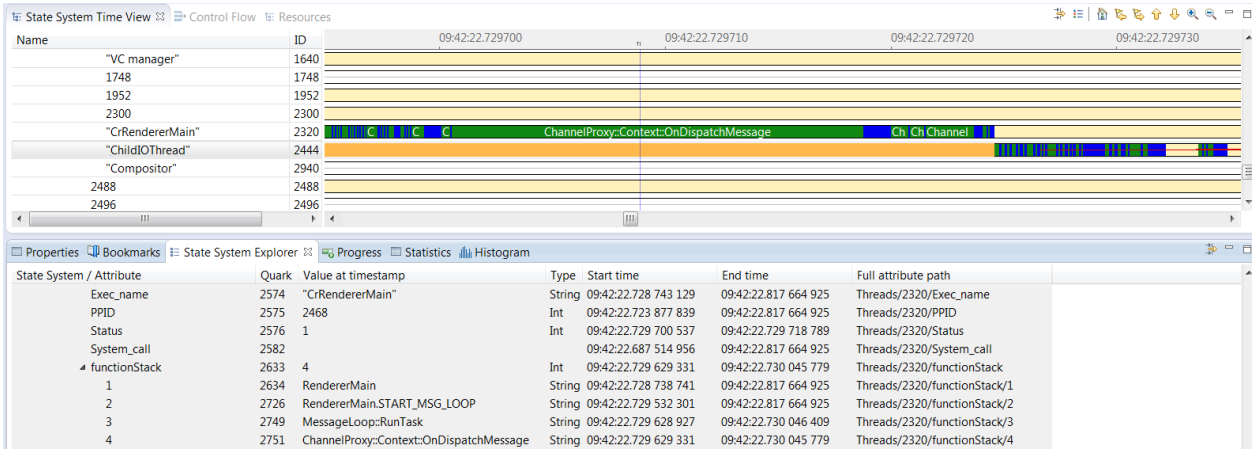

Figure 4.7 UST instrumentation combined with kernel events.

Virtual Machines monitoring

Another new application is the instrumentation of a physical server with virtual machines. The host and virtual machines are computers that can be traced independently with the kernel model. However, there is a potential gain of information by grouping together the different traces.

A simple example is to add the information of the CPU resources of the host in the virtual machine. We can then know if the virtual machine is running or preempted. This information is added to the model of the virtual machine. In this case, we see if the threads, thought to be running on the virtual CPUs, really have access to the physical CPUs, or if they are preempted.

Combining multiple kernel traces together presents a new challenge for the state system. Indeed, the number of attributes increases very rapidly with the number of virtual machines. Moreover, there query time cost in performance increases linearly with the number of attributes, as shown in Figure 4.8. This is why it is necessary to consider how to split the model into several separate state system trees.

The attribute tree is divided into groups of attributes (examples : CPUs, Threads, Files...). In the case of virtual machines monitoring, we use the name of the computer node as the first

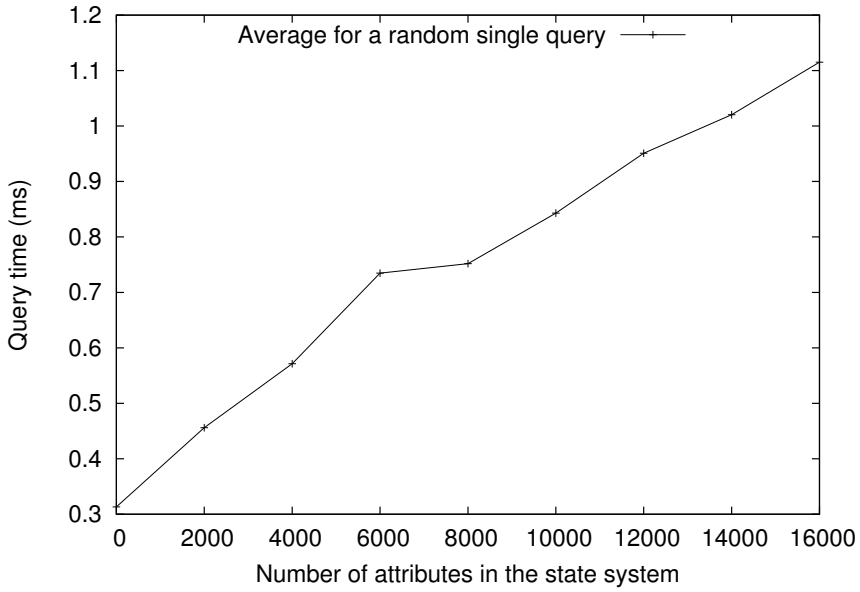

Figure 4.8 Average time to query a random interval in function of the number of different attributes in the state system.

level (see Figure 4.9). Then, each group may be stored in a separate state system tree to divide the problem size. A strategy to define folders as `mount` points, like in POSIX filesystems, could easily be added in the XML state provider header. This would allow the user to choose the backend used for each subfolder in the state system.

4.6.3 Queries optimization

Another interesting performance problem is with queries for the views. In order to have a good performance level, it is essential to minimize the number of queries in the system state. Several query types are implemented, and the performance depends on the information that we want to display. We detail a few use cases in this section.

Complete Query

The most expensive case is when we try to put bookmarks in places where we have detected an anomaly. Since it is necessary to check all state intervals for an attribute, possibly scanning the whole state history, these queries can be long, especially if the attribute often changes its state. For example, CPUs change state many times per second in a kernel trace. Initially, we do not have much information on the nature of the filter, so we cannot easily predict the time needed to get the information.

This approach is not typically used in the default user interface. If the user knows in ad-

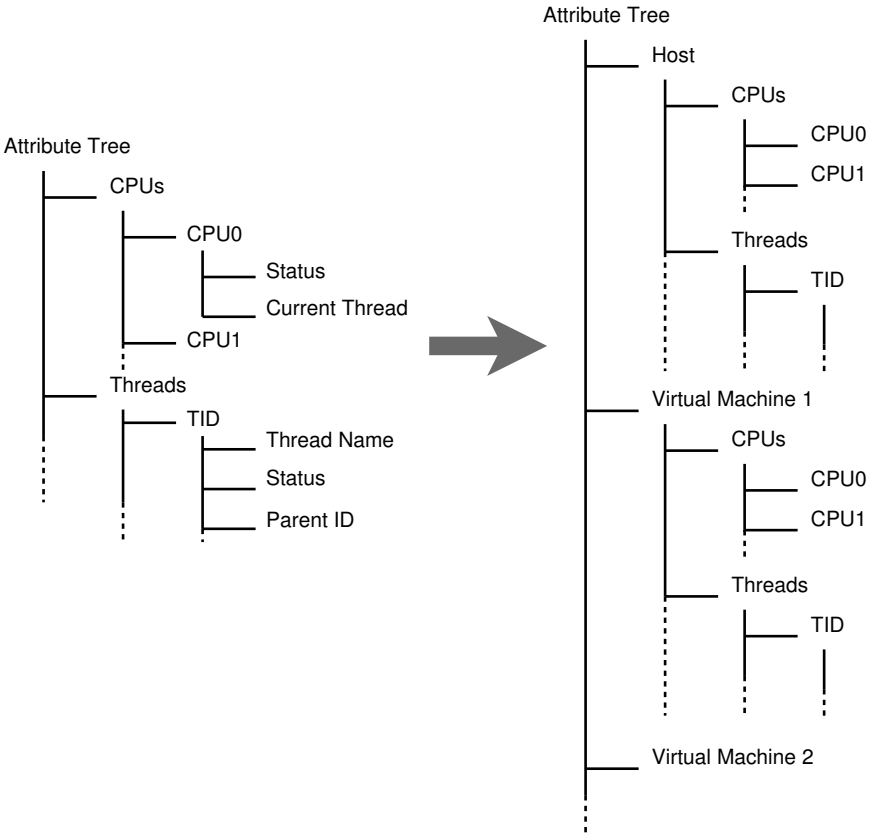

Figure 4.9 Adaptation of the structure of the attribute tree for the VM's application.

vance which filter it needs, it is possible to integrate the filter into the state system construction, with a small overhead. Then, the filter results will be available quickly in the user interface.

Resolution Query

Another optimization is the strategy implemented to quickly populate the Gantt chart view. This query adds the notion of resolution, because it's not possible, or necessary, to display more information than the number of available pixels to populate a view. Thus, the number of queries made in the state system will depend on the width of the view in pixels. There will be one interval queried per pixel, other intervals are ignored. This strategy is used to have a quick and significant overview of the trace at low zoom without querying the complete state system.

In the same way, if we want to make statistics by filtering, it is not necessary to use all state intervals to apply this filter. We can define a sampling strategy to estimate characteristics of the whole interval. For example, to display a bar chart with the thread usage (UST time,

Kernel time, Blocked time and Waiting time), we can query a fixed number of state intervals, and define a confidence interval for this metric.

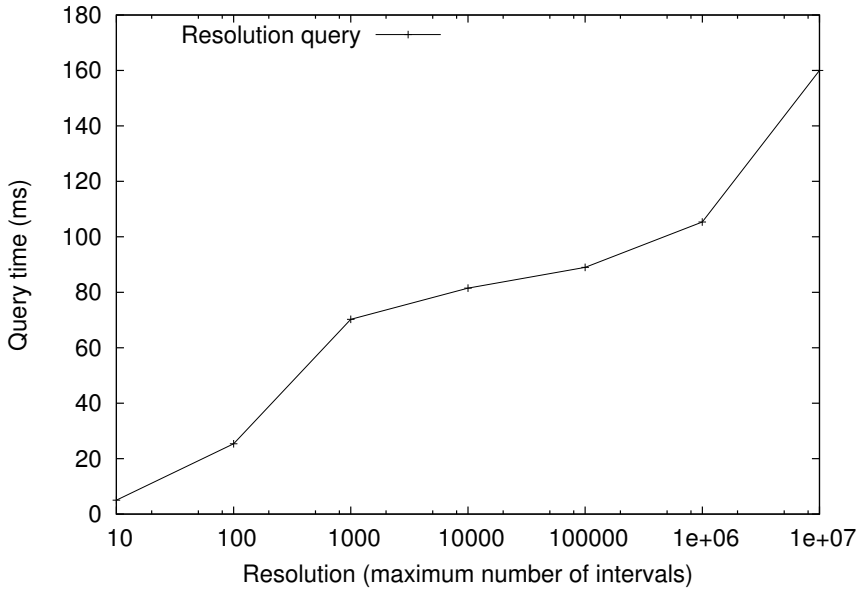

Figure 4.10 Average query time in function of the maximum number of intervals.

This optimization gives a logarithmic performance gain for the query time, depending on the maximum number of intervals we want to query, see Figure 4.10. The resolution provides information for every iteration step. However, if the state interval is longer than the iteration step, we don't make a query for each step. This way, on average, we have a logarithmic gain, as shown in Figure 4.10.

Partial query with a time Range

The last case is when the user wants to display the results of the filter (virtual states) in the Gantt chart view to highlight certain sections. A possible optimization is to only calculate the displayed time range. This technique is very responsive and is already used to populate the Gantt chart view when we use the zoom.

This query type is very interesting to reduce the query time. We have a linear improvement of the query time performance, see Table 4.3 and Figure 4.11. In addition, it is possible to combine this optimization with the previous one.

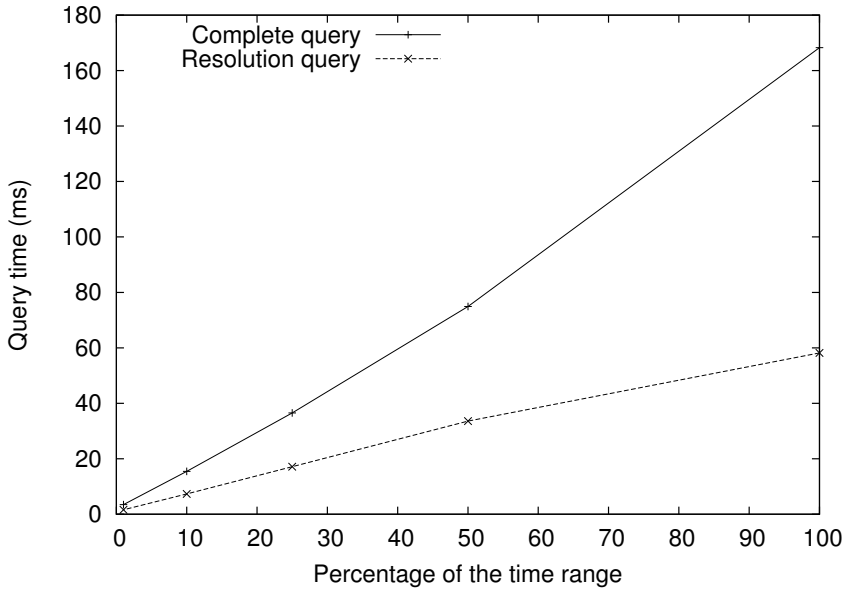

Figure 4.11 Average query time in function of the the time range percentage.

Tableau 4.3 Average query time in function of the the time range percentage.

Query	Complete (ms)	Resolution (ms)
Full range	168.3	57.38
50% range	74.90	33.60
25% range	36.52	17.16
10% range	15.44	7.32
1 % range	3.46	1.62

4.7 Conclusion

There is a tremendous amount of data available in traces and logs. However, it remains difficult for the developer or the system administrator to extract the right information to find the causes of his problems. Trace analysis software and trace viewers help to have meaningful graphical representations, but these representations are often generic and not very adaptable to specific contexts.

In this paper, we have presented a new tool architecture based on a generic declarative specification, a state history tree, and views. This allows the developer to put his knowledge of the product directly inside a model that can be used by the viewer to display synthetic information.

We have shown throughout this paper many successful applications of this proposed architecture. This work has generalized the way to model the state information of a system.

It is now possible to obtain a detailed view of the operating system internals for different operating systems of completely unrelated origin, like Linux and Windows. In addition, we demonstrated the use of this syntax to model more complex systems with multi-level traces, by combining User-space tracing and Kernel tracing, or Kernel tracing in several virtual and physical computers.

However, the possibilities are even greater. This declarative specification describes generic states and events. We can use it to create models with network events, telephony servers, financial records, etc. Moreover, the XML syntax is extensible. Thereby, the next step is to add more features, like critical path analysis. Another possibility for future work is to define dependencies between state changes, to have a more parallel construction of the state history tree.

CHAPITRE 5

DISCUSSION GÉNÉRALE

Ce chapitre revient sur les résultats présentés au chapitre 4. Certains aspects de ces résultats seront analysés plus en détail en donnant des exemples d'analyses qui ont pu être réalisées avec ce nouveau langage déclaratif.

5.1 Migration du modèle noyau Linux vers Windows

Cette partie présente les résultats qui ont pu être accomplis avec le système d'exploitation Microsoft Windows. Nous allons voir dans un premier temps les étapes qui ont été nécessaires pour convertir des événements dans le format approprié afin de pouvoir les lire dans TMF. Ensuite, nous détaillerons quelles sont les étapes importantes pour passer le modèle d'un traceur à l'autre.

5.1.1 Conversion des événements ETW vers CTF

Le traceur ETW du système d'exploitation de Microsoft fournit des événements dans un format particulier. Ce format n'étant pas libre, il n'est pas possible de concevoir un lecteur directement intégré à TMF.

Cependant, Microsoft fournit une API en C++ permettant de lire un fichier de trace et de retourner les événements sous la forme d'objets. Il a donc été nécessaire de créer un convertisseur permettant de transposer la trace dans un format compatible avec TMF. Le format le plus adapté pour être ouvert par TMF est le CTF, un format de trace libre et utilisé par LTTng. Un convertisseur permettant de passer du format d'ETW vers le CTF a été réalisé pour notre étude [1].

Ce convertisseur permet de transcrire tous les événements provenant d'une trace ETW. De plus, il permet d'utiliser les API de Microsoft pour faire de la résolution de symboles à partir du format PDB, ce qui permet de retrouver les noms des fonctions à partir de leur adresse.

Pour le moment, le convertisseur se limite à convertir un fichier de trace ETW en un fichier de trace CTF. Cependant, il est possible avec l'API de ETW de lancer directement le traçage à partir du convertisseur et de récupérer directement les événements du traceur sans

1. `https://github.com/fwininger/ETW2CTF`

passer par l'intermédiaire du format de sortie de trace ETW. Ce principe permettra dans le futur d'aller vers une solution permettant la visualisation en direct de la trace.

5.1.2 Équivalence des événements nécessaires au modèle

La seconde étape a été d'identifier les événements minimaux nécessaires pour produire une vue montrant les ressources utilisées sous la forme d'un Gantt.

L'élément clé pour cette modélisation est le suivi de l'ordonnanceur du système d'exploitation. En suivant les événements produits par l'ordonnanceur, il est possible de savoir quels processeurs et quels fils d'exécution sont actifs à un instant donné.

Pour cela, avec le modèle Linux, nous utilisons l'événement `sched_switch` qui contient les champs suivant : `prev_comm, prev_tid, prev_prio, prev_state, next_comm, next_tid et next_prio`. Cet événement permet de connaître le numéro et le nom du fil d'exécution qui s'exécutait sur un processeur, ainsi que les informations de celui qui va commencer à s'exécuter. Celui-ci peut éventuellement être `0` qui correspond au processus inactif, l'*idle*. Il est ainsi possible de définir les changements d'états permettant de modifier les attributs "*fil d'exécution courant du processeur n*", "*état du processeur n*" et "*état du fils d'exécution m*".

L'implémentation d'ETW fournit également un événement permettant de suivre les changements de contexte, l'événement `CSwitch`. Le tableau 5.1 montre les similitudes des deux événements.

Tableau 5.1 Similitudes entre les événements de changements de contexte Linux et Windows.

Champs de l'événement	sched_switch (Linux)	CSwitch (Windows)
Ancien Thread ID	prev_tid	OldThreadId
Nouveau Thread ID	next_tid	NewThreadId
Ancienne priorité	prev_prio	OldThreadPriority
Nouvelle priorité	next_prio	NewThreadPriority
État précédent	prev_state	PreviousCState

En plus de connaître le changement de fil d'exécution actif sur le système, l'événement fournit l'état du fil d'exécution précédent. Cette information permet de savoir s'il a été préempté ou si son travail est terminé pour l'instant. Dans le cas d'ETW, un champ supplémentaire, `OldThreadWaitReason`, permet de connaître l'état de sortie de l'ancien fil d'exécution.

De façon analogue, chaque partie du modèle du noyau Linux a été convertie avec les événements du noyau Windows. Le tableau 5.2 montre la correspondance entre les événements principaux du modèle. D'autres événements peuvent être ajoutés, comme les interruptions

(*IRQ*). Ces événements enrichissent l'information du modèle mais ne sont pas nécessaires à son fonctionnement.

Tableau 5.2 Correspondance des événements Linux et Windows pour le modèle noyau.

Événement	Linux	Windows
Changement de contexte	sched_switch	CSwitch
Début d'un appel système	sys_***	SysClEnter
Fin d'un appel système	exit_syscall	SysClExit
Début d'un nouveau processus	sched_process_fork	Start
Capture de l'état courant	lttng_statedump_process_state	DCStart, DCEnd

Les deux systèmes de traçage ne fonctionnement pas de la même manière pour la capture de l'état courant au moment de l'initialisation du traçage. ETW fournit un état complet du système au début et à la fin de la trace permettant d'avoir l'information pour reconstituer l'état de la machine. LTTng fournit quant à lui un état du système peu de temps après le début de la trace. Il peut y avoir cependant des événements qui précèdent la capture de l'état. L'initialisation du gestionnaire d'état n'étant pas faite, il peut y avoir une petite partie de la vue qui n'est pas correcte par rapport à la réalité. Par exemple, on n'est pas capable de savoir quel fil d'exécution est actif sur le système avant d'avoir eu un événement de l'ordonnanceur.

Malgré les différences de comportement des deux traceurs, il a été possible de montrer dans cette étude que le nouveau langage dédié offre une flexibilité suffisante pour changer de traceur ou de système d'exploitation. Dans le chapitre 4, nous avons vu un exemple de test qui a pu être tracé et visualisé dans le même logiciel à partir de traces recueillies sous Linux et Windows.

5.2 Modèle multi-niveaux

Une approche intéressante pour créer des analyses plus complètes est de tracer sur plusieurs "niveaux" d'application afin de synthétiser l'information issue de plusieurs couches d'abstraction. Cette approche a pour but de montrer qu'il est possible d'utiliser des événements provenant de sources différentes pour bâtir une analyse unique basée sur un gestionnaire d'état.

Afin de réaliser cet objectif, nous avons utilisé un logiciel libre, déjà instrumenté de façon optimale. Ce logiciel est le navigateur Google Chromium [2]. Plutôt que d'instrumenter chaque entrée et sortie de fonction, les points de traces de Chromium ont été placés de manière à avoir

2. `http://www.chromium.org/`

le début et la fin de chaque tâche. Une tâche désigne un ensemble d'instructions exécutant une fonctionnalité dans le navigateur. Ce système permet de limiter le nombre de points de trace utilisés afin de tracer en minimisant l'impact sur la performance du navigateur.

L'objectif de cette partie est de montrer qu'il est possible d'intégrer des événements de l'espace utilisateur dans le modèle noyau existant, afin d'utiliser les vues disponibles dans TMF pour trouver des nouvelles anomalies.

5.2.1 Enrichissement du modèle noyau

En prenant appui sur le modèle noyau de la partie précédente, nous souhaitons ajouter ici les informations des tâches de Chromium. Pour cela, nous disposons de trois nouveaux types d'événements : `Chrome_Begin` et `Chrome_End` qui annoncent respectivement le début et la fin d'une tâche, et `Chrome_Instant` qui diffuse des informations atemporelles telles que le nom des fils d'exécution de Chromium.

Nous savons que les tâches ne peuvent pas être déplacées d'un fil d'exécution à l'autre. Le modèle utilisé pour ajouter l'information repose donc sur l'utilisation d'une pile de tâches propre à chaque fil d'exécution. Pour cela, il suffit d'empiler le nom de la tâche dans un attribut lors de l'événement `Chrome_Begin` et de dépiler lors de l'événement `Chrome_End`.

Dans la syntaxe du langage dédié développé, cela revient à ajouter les deux changements d'états suivants au modèle qui utilise l'attribut :

```
/Thread/${event/context.tid}/functionStack
```

```
<eventHandler eventname="Chrome_Begin">
    <stateChange>
        <attribute constant="Threads" />
        <attribute eventfield="context.tid" />
        <attribute constant="functionStack" />
        <value stack="push" eventfield="name" />
    </stateChange>
</eventHandler>

<eventHandler eventname="Chrome_End">
    <stateChange>
        <attribute constant="Threads" />
        <attribute eventfield="context.tid" />
        <attribute constant="functionStack" />
        <value stack="pop" />
```

```
    </stateChange>
</eventHandler>
```

La pile d'exécution des tâches est alors visualisable pour chaque fil d'exécution. De même, la tâche courante, c'est-à-dire le haut de la pile, est affichable comme un label dans l'état qui correspond à l'exécution en espace utilisateur. La Figure 5.1 montre l'exemple de la fonction `ChannelProxy::Context::OnDispatchMessage` qui est active sur la portion verte correspondant à l'état de l'espace utilisateur. Une seconde vue permet d'explorer à un temps donné la pile complète contenue dans le gestionnaire d'état.

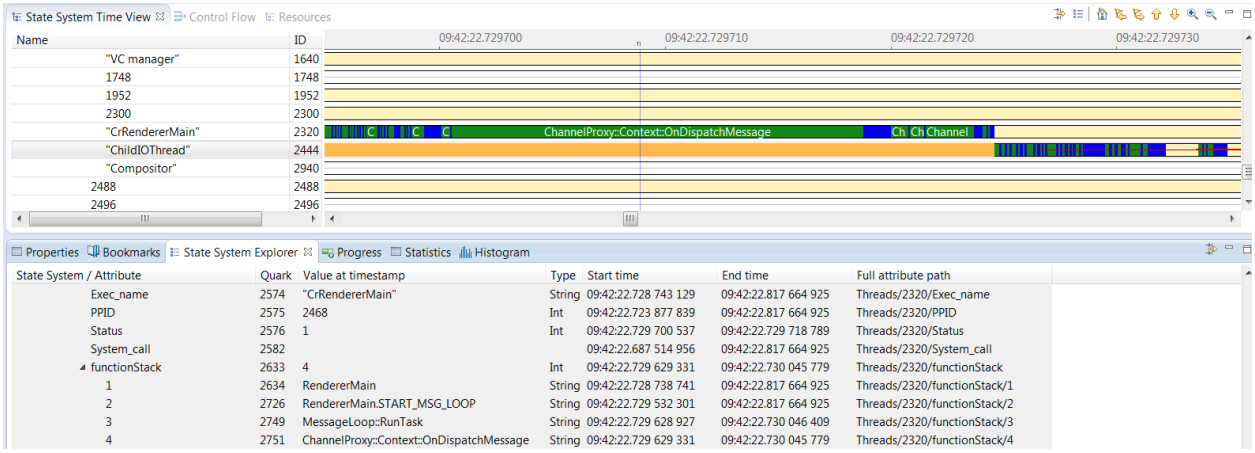

Figure 5.1 Vue dans TMF représentant l'exécution de Chrome avec les échanges de messages.

L'architecture de Chromium est très fortement multiprocessus et multithreads. Son objectif est notamment de garantir la sécurité des utilisateurs avec un procédé de bac à sable, ou *Sandbox*, en séparant tous les onglets dans des processus indépendants. Les tâches à effectuer pour le fonctionnement du navigateur sont coordonnées par l'envoi de messages entre les différents fils d'exécution. Il est possible de suivre l'envoi et la réception des messages avec les informations contenues dans les événements de Chromium. Ces informations passent par l'exécution de la fonction `MessageLoop::PostTask`.

Afin de rendre visible l'envoi de messages dans la vue de TMF, il est possible de filtrer cette fonction spécifique. Un identificateur unique est également fourni avec cette fonction pour pouvoir corréler la demande d'exécution de la tâche avec la réponse du fil d'exécution qui va la traiter.

Le gestionnaire d'état de TMF est utilisé pour stocker l'information. Pour cela, une branche `/Task/ID` a été ajoutée. Chaque attribut contient deux états. Le premier permet d'avoir le début de l'envoi du message et contient le numéro du fil d'exécution qui a fait la

demande de la tâche. Le second débute lorsque la tâche démarre et contient le numéro du fil d'exécution qui effectue le travail de la tâche. Ces informations permettent de définir une flèche montrant la communication entre l'émetteur et le destinataire du message.

Ce principe fonctionne comme le montre la Figure 5.2. Les flèches rouges correspondent aux envois de messages d'un fil d'exécution à l'autre. Cependant, il est nécessaire de créer un attribut par tâche exécutée. Cette contrainte n'est pas compatible avec une utilisation optimale du système de sauvegarde du gestionnaire d'état, comme nous allons le voir dans le paragraphe 6.2.

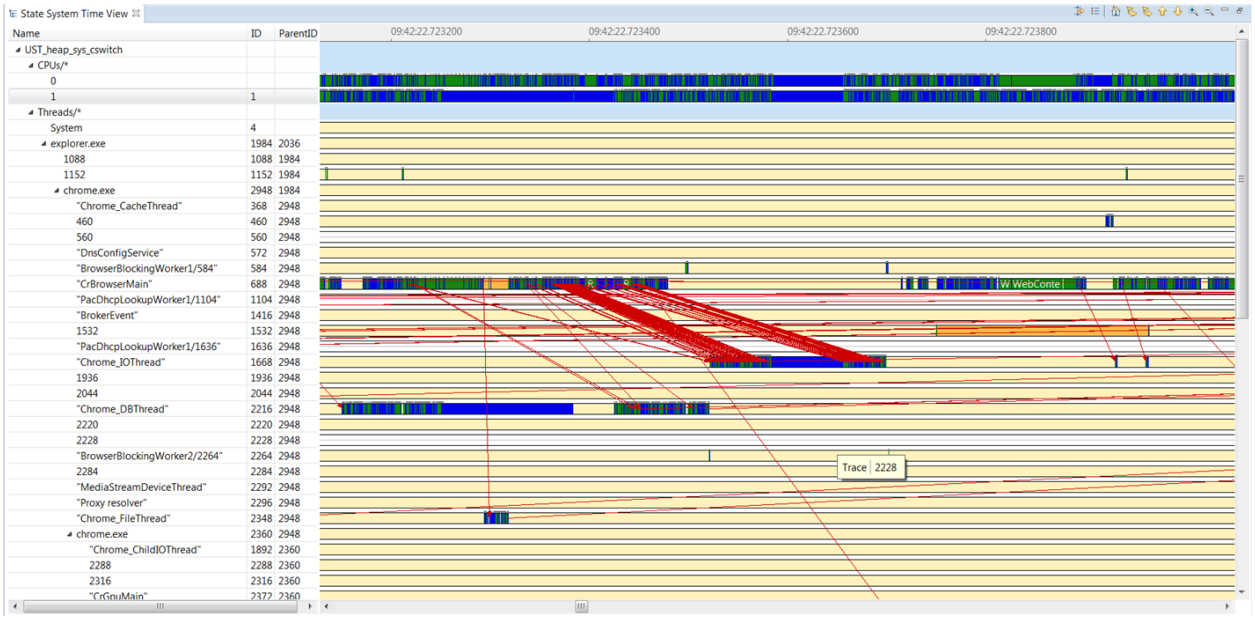

Figure 5.2 Vue dans TMF représentant l'exécution de Chrome avec les échanges de messages.

La figure 5.2 montre les interactions entre le fil d'exécution principal et les fils d'exécution effectuant les tâches secondaires comme les entrées/sorties ou l'accès à la base de données.

5.2.2 Détection d'anomalies

L'objectif de l'étude est de montrer qu'il est possible de détecter de nouveaux problèmes qui ne peuvent pas être visualisés avec les outils existants.

L'avantage de notre approche par rapport aux traces disponibles à l'intérieur de Chromium est que l'on peut utiliser les événements du noyau du système d'exploitation pour ajouter de l'information supplémentaire sur les appels systèmes et l'ordonnancement des fils d'exécution.

Comme nous l'avons vu précédemment, l'architecture de Chromium est fortement parallèle. Il y a plus d'une dizaine de fils d'exécution par processus et chaque onglet est dans un processus séparé. Cependant, sur un ordinateur, il y a au maximum un fil d'exécution qui est actif par processeur (éventuellement virtuel), soit un nombre réduit par rapport aux nombres de fils d'exécution de Chromium. Les événements du noyau nous permettent de connaître les fils d'exécution actifs à chaque instant de la trace. Il est alors possible de savoir si certains sont préemptés par d'autres.

Cette information n'est pas disponible au niveau de l'application. Ainsi, si on se limite aux événements de Chromium, on peut penser qu'une tâche s'exécute entre son événement de début et de fin, alors qu'elle a été préemptée en cours d'exécution. Cela peut induire de mauvaises interprétations des résultats.

Nous allons maintenant voir un type de problème dans l'utilisation de Chromium qu'il a été possible d'identifier à l'aide de notre nouvelle analyse. La trace recueillie pour l'analyse a été faite à l'aide d'une machine virtuelle avec deux processeurs virtuels. Il ne peut donc y avoir que deux fils d'exécution actifs simultanément.

Le comportement souhaité du navigateur Chromium est de ne jamais être bloqué par des appels systèmes pour des tâches telles que des lectures ou écritures sur le disque dur. Pour avoir ce comportement, le fil d'exécution principal "CrRenderMain" exécute les tâches d'entrée/sortie sur un fil d'exécution dédié "Chrome_ChildIOThread" par un envoi de messages. Ce fil d'exécution est moins prioritaire que le fil d'exécution principal.

Ainsi, le comportement normal est modélisé dans la Figure 5.3. On voit en vert (application) et en bleu (noyau) lorsque les fils d'exécution sont actifs, en jaune lorsqu'ils sont en attente sans travail et en orange lorsqu'ils ont été préemptés. Dans cette représentation, le fil d'exécution principal a la priorité sur celui qui gère les entrées/sorties. Ce dernier peut s'exécuter uniquement lorsque l'autre n'a plus de travail. Dans le cas contraire, il se fait préempter.

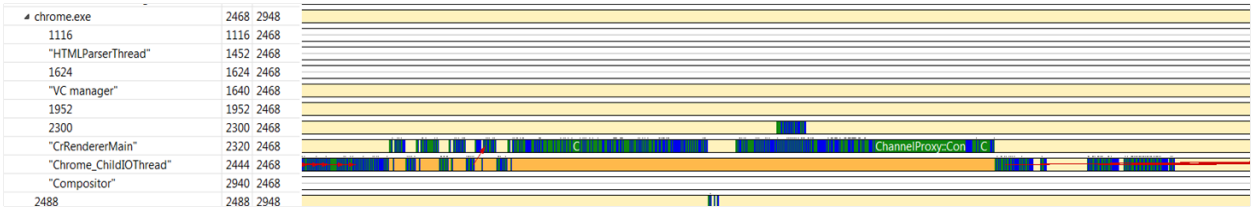

Figure 5.3 Comportement normal des fils d'exécution de Chromium.

Cependant, dans certaines conditions, le comportement de Chromium n'est pas celui qui

est attendu. La Figure 5.4 montre une exécution dans laquelle chaque fois que le fil d'exécution principal envoie un message pour lancer une tâche, il se fait préempter par celui qui s'occupe des entrées/sorties.

Ce type de comportement est très pénalisant pour l'interface utilisateur. Même s'il n'a pas d'impact sur le temps de calcul global du logiciel, ce comportement ralentit l'affichage et donc influe négativement sur la perception de fluidité de l'utilisateur.

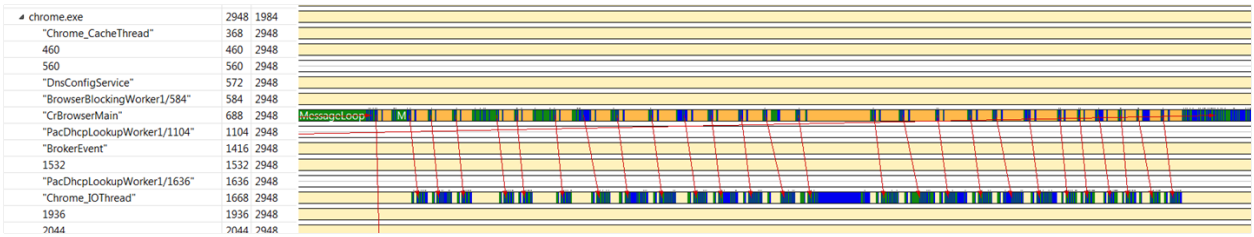

Figure 5.4 Comportement problématique des fils d'exécution de Chromium.

Il est facile de détecter automatiquement ce type de problèmes par l'utilisation de filtres sur les états. Il suffit de spécifier une condition de la forme : si `CrRenderMain == Préempté` et `Chrome_ChildIOThread == Actif` alors `Problème`.

Nous avons pu voir dans cette section l'intérêt de combiner ensemble des traces de différentes origines pour concevoir un modèle à états complet. Non seulement il est possible d'avoir une meilleure représentation du système à partir d'une analyse dans le visualiseur, mais il est également possible de définir sur les états bien choisis des filtres pour détecter de nouveaux problèmes de comportement.

Nous allons maintenant voir une limitation qui a été rencontrée avec une solution qui a pu être apportée.

5.3 Limitation du nombre d'attributs

Grâce à la flexibilité introduite avec le langage déclaratif, il est devenu plus facile de concevoir des modèles d'analyse et d'y ajouter des informations. Ainsi, le nombre toujours plus important de paramètres (attributs) surveillés dans le système a permis de mettre en évidence de nouvelles limites du gestionnaire d'état.

Auparavant, de nombreux tests de performance ont été réalisés pour mesurer l'impact de la taille de la trace sur la durée des requêtes (voir Montplaisir-Goncalve, 2011). Cependant, dans notre cas, nous abordons le problème sous un autre angle, celui de la densité d'information.

Ce problème, qui n'est pas visible avec le modèle des traces noyau Linux, se révéla très contraignant pour le modèle regroupant les traces noyaux Windows avec celles de Chromium, conduisant à des ralentissements inacceptables.

5.3.1 Identification du problème

Le composant de TMF permettant de sauvegarder sur disque les intervalles produits par le gestionnaire d'état se nomme l'arbre à historique. Il a été présenté par Montplaisir-Goncalves *et al.* (2013) comme une structure efficace pour sauvegarder des intervalles. Grâce à une structure d'arbre, l'arbre à historique est capable d'effectuer les requêtes dans un temps logarithmique par rapport au nombre d'intervalles d'état présents.

Afin de garantir cette performance tout en ayant un temps de construction rapide, l'arbre à historique fait l'hypothèse que les intervalles arrivent triés par temps de fin. Cette propriété permet d'éviter le rebalancement de l'arbre durant sa construction.

L'arbre est composé de noeuds qui sont des conteneurs d'intervalles. Chaque noeud peut contenir 64 Ko d'intervalles, ce qui correspond à environ 2600 intervalles. De plus, chaque noeud peut avoir jusqu'à 50 enfants. Lorsqu'un noeud est plein, il est sauvegardé sur le disque dur avec tous ses enfants. Un autre noeud frère est créé pour contenir des intervalles.

La Figure 5.5 montre que le sous-arbre 1 contient les intervalles dont les temps sont entre 0 et 20. Lorsque ce sous-arbre est plein, le sous-arbre 2 est créé pour les intervalles commençant à partir de 21. Ce système permet de maintenir seulement la branche la plus à droite en mémoire.

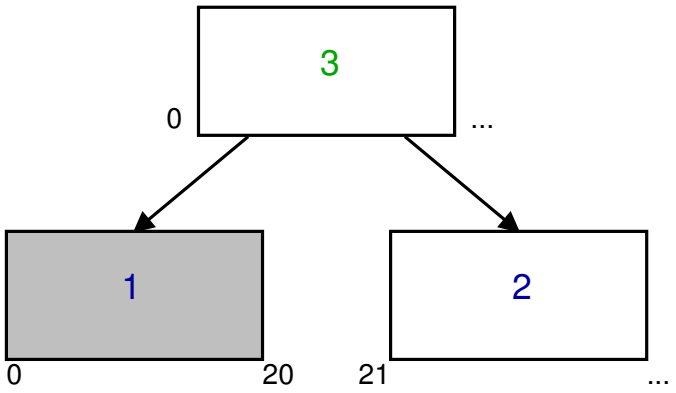

Figure 5.5 Construction de l'historique.

L'inconvénient de ce procédé est qu'une discontinuité est créée à la fermeture de la branche 1. En effet, on souhaite commencer à remplir en priorité le nouveau noeud 2 avec des intervalles commençant à partir de t+1 par rapport au noeud précédent, ici 21.

Cependant, au moment de la fermeture du noeud, il existe pour chaque attribut un intervalle commençant avant le temps t+1. Ces intervalles ne pourront pas être stockés dans le nouveau noeud 2 car celui-ci ne peut contenir que des intervalles commençant à partir de t+1. Ainsi ces intervalles vont se retrouver dans le noeud racine 3.

Le problème intervient lorsqu'il y a plus d'attributs que le nombre d'intervalles que peut contenir le noeud. Dans ce cas, le noeud racine 3 se remplit avant son enfant, ce qui a pour conséquence que l'arbre dégénère progressivement en une liste chainée.

La Figure 5.6 illustre se problème. Pour un arbre de 8192 attributs, les requêtes se font en moyenne à 22.39ms au lieu de moins de 1ms habituellement.

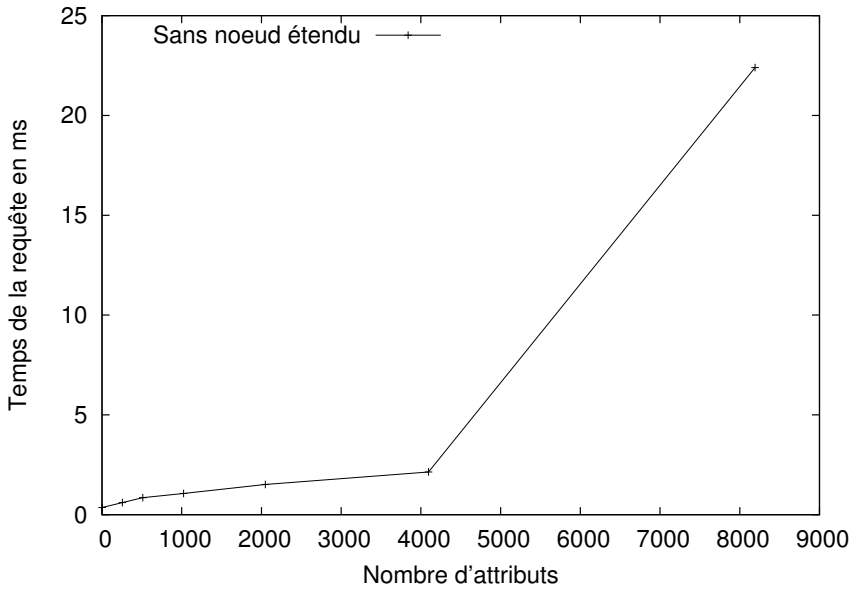

Figure 5.6 Temps moyen d'une requête dans l'historique sans noeud étendu.

Par ailleurs, il y a un nombre très important de noeuds vides qui sont créés. L'espace disque utilisé par l'arbre à historique est de 726Mo au lieu de 17.2Mo pour un arbre équilibré contenant le même nombre d'intervalles.

5.3.2 Noeuds étendus

La solution proposée pour rééquilibrer la structure de données est de créer des extensions aux noeuds. Au lieu d'être d'une taille fixée à 64Ko, les noeuds pourront comporter des extensions d'une taille multiple de la taille initiale.

Le nouvel algorithme mis en place pour le remplissage des noeuds impose que chaque noeud doit avoir au moins trois enfants avant d'être plein. Tant qu'il n'y a pas au moins 3 enfants, des extensions sont créées au noeud. Cela assure d'avoir en permanence une structure

d'arbre non dégénérée. De plus, avec cette propriété, on garantit une compacité des noeuds de l'arbre d'au moins 75%.

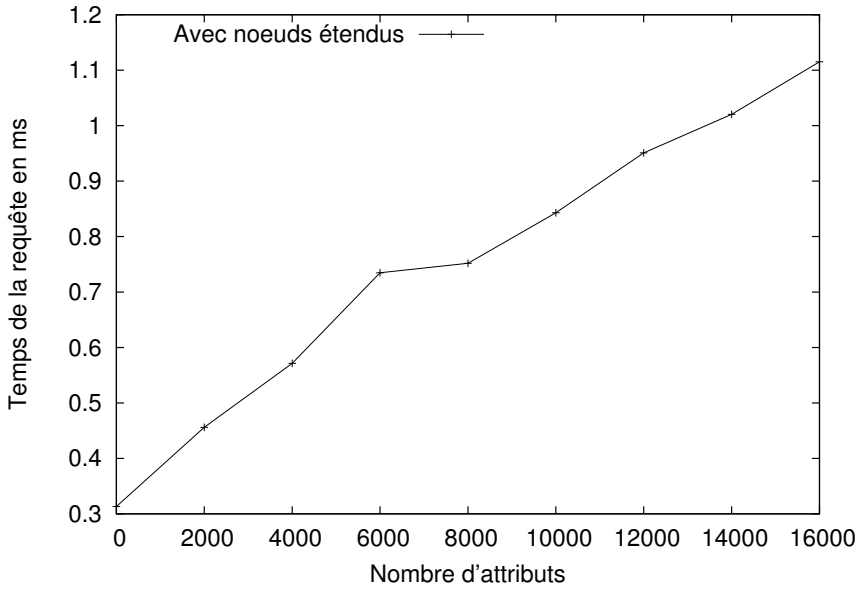

Figure 5.7 Temps moyen d'une requête dans l'historique avec noeuds étendus.

La Figure 5.7 montre les résultats corrigés des requêtes dans l'arbre à historique. Les résultats montrent qu'il n'y a plus de limite pour laquelle l'arbre à historique dégénère en liste.

Néanmoins, on observe que l'augmentation du nombre d'attributs dans le gestionnaire d'état augmente de façon linéaire le temps des requêtes. Cette caractéristique fera l'objet d'une discussion dans la section 6.2 pour les limitations du langage.

CHAPITRE 6

CONCLUSION ET RECOMMANDATIONS

Nous allons maintenant revenir sur l'ensemble des travaux présentés dans ce mémoire afin d'en faire la conclusion. Nous en profiterons pour aborder les limitations de la solution proposée et les améliorations futures qui pourront y être ajoutées.

6.1 Synthèse des travaux

Dans ce mémoire, nous avons étudié la problématique de l'introduction d'un langage déclaratif permettant d'exprimer les relations entre les événements contenus dans une trace système. L'objectif est de concevoir simplement de nouvelles analyses avec les outils existants et de résoudre de nouveaux problèmes de performance et de sécurité. Ces analyses reposent sur un gestionnaire d'état permettant de modéliser et de sauvegarder l'état du système à tout instant de la trace. Pour répondre à cet objectif, nous avons mis en place une méthodologie en trois étapes.

La première concerne l'expressivité du langage. Pour cela, nous avons caractérisé de manière formelle les changements d'états d'un système provoqués par les événements d'une trace. Cela a permis de définir les opérations nécessaires pour créer un gestionnaire d'état. Une fois l'expressivité du langage suffisante pour caractériser tous les types de trace, nous avons choisi d'utiliser un langage de balisage générique pour définir la grammaire associée au langage.

Deuxièmement, nous avons cherché à valider la performance du langage proposé. À cette fin, l'intégration de la solution dans TMF a permis de faire les mesures nécessaires pour montrer que l'implémentation du langage n'affecte pas de façon significative le fonctionnement du logiciel. Les résultats montrent que l'impact sur la performance est inférieur à l'écart-type entre les différentes itérations du test.

Enfin, la troisième étape de la méthodologie concerne l'utilisabilité. Grâce à la solution développée dans notre étude, nous avons pu créer diverses analyses. De la plus simple, réalisée en moins d'une heure par un nouvel utilisateur de l'outil, en utilisant seulement trois types d'événements et permettant de caractériser le fonctionnement des différentes instances d'un programme, à la plus complexe, fusionnant plusieurs types de traces. Nous avons pu montrer qu'il est possible d'abstraire suffisamment les modèles décrits avec le nouveau langage déclaratif pour le rendre indépendant du système d'exploitation. Ainsi, il a été possible de réaliser les mêmes analyses au niveau du noyau sous Linux ou Windows. Une autre piste

de flexibilité qui a été explorée est de créer un modèle utilisant différents types de traces. L'étude réalisée a montré l'avantage d'avoir des informations du noyau dans l'étude d'une trace d'une application utilisateur.

6.2 Limitations de la solution proposée

La solution proposée permet d'utiliser complètement le gestionnaire d'état existant. Elle agit comme une nouvelle couche d'abstraction pour créer de nouvelles analyses.

Une limitation du fonctionnement actuel de la construction du gestionnaire d'état est qu'elle est réalisée de manière séquentielle. En effet, il n'est pas possible d'aller à n'importe quel point de la trace, puisque la construction se base sur la lecture continue de la liste d'événements.

Comme nous l'avons mentionné dans l'étude, le gestionnaire d'état peut parfois contenir des informations corrompues suite à une mauvaise initialisation ou à une perte d'événements. La possibilité d'avancer ou de reculer à un point de la trace permettrait de corriger certaines informations manquantes au prix d'un coût supplémentaire pour la construction du gestionnaire d'état.

L'ajout du branchement conditionnel au niveau de notre langage permettrait de le rendre complet au sens de Turing, ce qui garantirait la possibilité de résoudre n'importe quel algorithme à partir de notre langage combiné à la mémoire du gestionnaire d'état. L'absence de saut conditionnel représente certes une limite pour notre langage, mais cette solution a tout de même été retenue, car elle permet d'optimiser les performances en lecture de la trace sur le disque dur et assure d'avoir un nombre fini d'opérations nécessaires pour la construction du gestionnaire d'état.

Une seconde limitation de la solution proposée est qu'elle ne permet pas de définir les dépendances entre les différents changements d'états. Afin de rendre la construction du gestionnaire d'état parallèle ou de pouvoir démarrer la construction à un point spécifique de la trace, il serait nécessaire de connaître les informations des dépendances entre les différents états créés. Certains événements n'agissent en effet que sur un groupe d'attributs spécifiques du gestionnaire d'état. Prenons l'exemple des tâches dans Chromium. Les changements d'états générés par les événements de ce navigateur n'ont aucun impact sur les changements d'états générés par les événements de la trace noyau. Il serait donc possible d'effectuer la construction de ces deux parties de l'analyse de façon parallèle.

6.3 Améliorations futures

Les travaux de ce mémoire ouvrent de nouvelles possibilités pour TMF en permettant à un utilisateur plutôt qu'à un développeur de définir des analyses spécifiques.

Nous avons introduit une syntaxe permettant d'ajouter et d'échanger de nouvelles analyses basées sur une machine à états. En plus des travaux spécifiques sur la définition du modèle à base d'états, l'extensibilité de l'XML permet de spécifier des paramètres d'affichage pour les vues du logiciel d'analyse.

Nos travaux ont montré qu'il est possible d'utiliser les vues génériques telles que les diagrammes de Gantt pour afficher l'évolution d'attributs spécifiés à l'aide d'un fichier XML. Pour le futur, une amélioration possible serait d'effectuer ce travail pour toutes les vues de TMF afin de les rendre disponibles pour les nouvelles analyses spécifiques.

Enfin, l'objectif de la solution proposée est d'avoir un ficher d'interface suivant une grammaire spécifique pour la représentation d'analyses à base d'état. Pour le moment, seule la partie traitement du fichier a été implémentée. Une amélioration future intéressante serait de créer une interface graphique permettant de construire intéractivement le modèle à états et de générer un fichier XML qui pourra être utilisé par une analyse de TMF.

RÉFÉRENCES

ANDROID PROJECT (2014). Analyzing display and performance in android. `http://developer.android.com/tools/debugging/systrace.html`. Consulté en février 2014.

BALIMA, D. (2011). Awk : le langage script de référence pour le traitement de fichiers. `http://www.unixgarden.com/index.php/gnu-linux-magazine/`. Consulté en février 2014.

BLUNCK, J., DESNOYERS, M. et FOURNIER, P.-M. (2009). Userspace application tracing with markers and tracepoints. *Proceedings of the Linux Kongress*.

BOLOUR, A. (2003). Notes on the eclipse plug-in architecture. `eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html`. Consulté en février 2014.

BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C. M., MALER, E. et YERGEAU, F. (1997). Extensible markup language (xml). *World Wide Web Journal*, 2, 27–66.

CANTRILL, B., SHAPIRO, M. W., LEVENTHAL, A. H. *ET AL.* (2004). Dynamic instrumentation of production systems. *USENIX Annual Technical Conference, General Track*. 15–28.

CHAN, A., GROPP, W. et LUSK, E. (2008). An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming*, 16, 155–165.

CHROMIUM PROJECT (2013a). Adding traces to chromium/webkit/javascript. `http://www.chromium.org/developers/how-tos/trace-event-profiling-tool`. Consulté en février 2014.

CHROMIUM PROJECT (2013b). Trace event profiling tool. `http://www.chromium.org/developers/how-tos/trace-event-profiling-tool`. Consulté en février 2014.

CHROMIUM PROJECT (2014). Design documents. `http://www.chromium.org/developers/design-documents`. Consulté en février 2014.

COHEN, I., GOLDSZMIDT, M., KELLY, T., SYMONS, J. et CHASE, J. S. (2004). Correlating instrumentation data to system states : a building block for automated diagnosis and control. *Proceedings of the 6th conference on Symposium on Operating Systems Design Implementation -Volume 6*. USENIX Association, Berkeley, CA, USA, 16–16.

COHEN, I., ZHANG, S., GOLDSZMIDT, M., SYMONS, J., KELLY, T. et FOX, A. (2005). Capturing, indexing, clustering, and retrieving system history. *SIGOPS Operating Systems Review*, 39, 105–118.

CORBET, J. (2008). Tracing : no shortage of options. http://lwn.net/Articles/291091/. Consulté en février 2014.

DESCHÊNES, J.-H., DESNOYERS, M. et DAGENAIS, M. R. (2008). Tracing time operating system state determination. *Open Software Engineering Journal*, 2, 40–44.

DESFOSSEZ, J. (2011). *Résolution de problème par suivi de métriques dans les systèmes virtualisés*. Mémoire de maîtrise, École Polytechnique de Montréal.

DESNOYERS, M. (2009). *Low-impact operating system tracing*. Thèse de doctorat, École Polytechnique de Montréal.

DESNOYERS, M. et DAGENAIS, M. (2008). Lttng : Tracing across execution layers, from the hypervisor to user-space. *Linux Symposium*.

DESNOYERS, M. et DAGENAIS, M. R. (2006). The lttng tracer : A low impact performance and behavior monitor for gnu/linux. *OLS (Ottawa Linux Symposium)*. 209–224.

DESNOYERS, M., DESFOSSEZ, J. et GOULET, D. (2012). Lttng 2.0 : Tracing for power users and developers - part 1. http://lwn.net/Articles/491510/. Consulté en février 2014.

DESNOYERS, M. et EFFICIOS INC (2013). Common trace format. http://www.efficios.com/ctf. Consulté en février 2014.

ECKMANN, S. T., VIGNA, G. et KEMMERER, R. A. (2002). Statl : An attack language for state-based intrusion detection. *Journal of Computer Security*, 10, 71–103.

ECLIPSE FOUNDATION (2014). Rich client platform. http://wiki.eclipse.org/Rich_Client_Platform. Consulté en février 2014.

EDGE, J. (2009). Perfcounters added to the mainline. http://lwn.net/Articles/339361/. Consulté en février 2014.

EIGLER, F. C. et RED HAT (2006). Problem solving with systemtap. *Proceedings of the Ottawa Linux Symposium*. 261–268.

EZZATI-JIVAN, N. et DAGENAIS, M. R. (2012). A stateful approach to generate synthetic events from kernel traces. *Advances in Software Engineering*, 2012, 6.

EZZATI-JIVAN, N. et DAGENAIS, M. R. (2013). A framework to compute statistics of system parameters from very large trace files. *ACM SIGOPS Operating Systems Review*, 47, 43–54.

FICHEUX, P. (2011). Introduction à ftrace, linux embedded. http://www.linuxembedded.fr/2011/03/introduction-a-ftrace/. Consulté en février 2014.

FOURNIER, P.-M., DESNOYERS, M. et DAGENAIS, M. R. (2009). Combined tracing of the kernel and applications with lttng. *Proceedings of the 2009 linux symposium*.

GIRALDEAU, F., DESFOSSEZ, J., GOULET, D., DAGENAIS, M. et DESNOYERS, M. (2011). Recovering system metrics from kernel trace. *OLS (Ottawa Linux Symposium)*. 109–116.

GOSWAMI, S. (2005). An introduction to kprobes. `http://lwn.net/Articles/132196/`. Consulté en février 2014.

GROPP, W. D., LUSK, E. L. et SKJELLUM, A. (1999). *Using MPI : portable parallel programming with the message-passing interface*, vol. 1. the MIT Press.

HABRA, N., LE CHARLIER, B., MOUNJI, A. et MATHIEU, I. (1992). Asax : Software architecture and rule-based language for universal audit trail analysis. *Computer Security—ESORICS 92*, Springer. 435–450.

KENISTON, J. et DRONAMRAJU, S. (2010). Uprobes : User-space probes. `http://events.linuxfoundation.org/slides/lfcs2010_keniston.pdf`. Consulté en février 2014.

LLOYD, J. W. (1994). Practical advantages of declarative programming. *Joint Conference on Declarative Programming, GULP-PRODE*. vol. 94, 94.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J. et HAZELWOOD, K. (2005). Pin : building customized program analysis tools with dynamic instrumentation. *Acm Sigplan Notices*. ACM, vol. 40, 190–200.

MATNI, G. (2009). *Detecting Problematic Execution Patterns Through Automatic Kernel Trace Analysis*. Mémoire de maîtrise, École Polytechnique de Montréal.

MATNI, G. et DAGENAIS, M. (2009). Automata-based approach for kernel trace analysis. *Electrical and Computer Engineering, 2009. CCECE'09. Canadian Conference on*. IEEE, 970–973.

MONTPLAISIR, A., EZZATI-JIVAN, N., WININGER, F. et DAGENAIS, M. (2013). Efficient model to query and visualize the system states extracted from trace data. *Runtime Verification*. Springer, 219–234.

MONTPLAISIR-GONCALVE, A. (2011). *Stockage sur Disque pour Accès Rapide d'Attributs avec Intervalles de Temps*. Mémoire de maîtrise, École Polytechnique de Montréal.

MONTPLAISIR-GONCALVES, A., EZZATI-JIVAN, N., WININGER, F. et DAGENAIS, M. (2013). State history tree : an incremental disk-based data structure for very large interval data. *2013 International Conference on Big Data*. IEEE, 716–724.

PARK, I. et BENDETOVERS, A. (2009a). Core os events in windows 7, part 1. `http://msdn.microsoft.com/fr-fr/magazine/ee412263.aspx`. Consulté en février 2014.

PARK, I. et BENDETOVERS, A. (2009b). Etw core instrumentation events in windows 7, part 2. http://msdn.microsoft.com/fr-fr/magazine/ee358703.aspx. Consulté en février 2014.

PARK, I. et BUCH, R. (2007). Improve debugging and performance tuning with etw. http://msdn.microsoft.com/en-us/magazine/cc163437.aspx. Consulté en février 2014.

PRASAD, V., COHEN, W., EIGLER, F., HUNT, M., KENISTON, J. et CHEN, B. (2005). Locating system problems using dynamic instrumentation. *2005 Ottawa Linux Symposium*. Citeseer, 49–64.

RAJADURAI, A. (2012). Observing and optimizing your application with dtrace. http://dtracehol.com. Consulté en février 2014.

RAPP, C. W. (2014). Smc : The state machine compiler. http://smc.sourceforge.net/. Consulté en février 2014.

ROESCH, M. *ET AL.* (1999). Snort : Lightweight intrusion detection for networks. *LISA*. vol. 99, 229–238.

ROSTEDT, S. (2008). ftrace - function tracer . https://www.kernel.org/doc/Documentation/trace/ftrace.txt. Consulté en février 2014.

ROSTEDT, S. (2010a). Using the trace_event() macro (part 1). http://lwn.net/Articles/379903/. Consulté en février 2014.

ROSTEDT, S. (2010b). Using the trace_event() macro (part 2). http://lwn.net/Articles/381064/. Consulté en février 2014.

ROSTEDT, S. (2010c). Using the trace_event() macro (part 3). http://lwn.net/Articles/383362/. Consulté en février 2014.

SCHNORR, L. M., HUARD, G. et NAVAUX, P. O. (2010). Triva : Interactive 3d visualization for performance analysis of parallel applications. *Future Generation Computer Systems*, 26, 348–358.

SCHNORR, L. M., HUARD, G. et NAVAUX, P. O. A. (2009). Towards visualization scalability through time intervals and hierarchical organization of monitoring data. *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, Washington, DC, USA, CCGRID 09, 428–435.

SECNOLOGY (2014). Website. http://www.secnology.com/. Consulté en février 2014.

THOMPSON, H. S. (2004). Xml schema part 1 : Structures second edition.

TURING, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 42, 230–265.

WALY, H. (2011). *Automated Fault Identification : Kernel Trace Analysis*. Mémoire de maîtrise, Université Laval.

ZAKI, O., LUSK, E., GROPP, W. et SWIDER, D. (1999). Toward scalable performance visualization with jumpshot. *International Journal of High Performance Computing Applications*, 13, 277–288.